A

Generate Collection | Print

L1: Entry 1 of 5 File: USPT Sep 17, 2002

US-PAT-NO: 6453356
DOCUMENT-IDENTIFIER: US 6453356 B1

TITLE: Data exchange system and method

DATE-ISSUED: September 17, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Sheard; Nicolas C.	Palo Alto	CA		
Fischer; Larry J.	Campbell	CA		
Matthews; Richard W.	Redwood City	CA		
Himabindu; Gurla	Sunnyvale	CA		
Hu; Qilin	Mountain View	CA		
Zheng; Wendy J.	Cupertino	CA		
Mow; Boyle Y.	Freemont	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
ADC Telecommunications, Inc.	Eden Prairie	MN			02

APPL-NO: 09/ 060667 [PALM]
DATE FILED: April 15, 1998

INT-CL: [07] G06 F 15/16

US-CL-ISSUED: 709/231
US-CL-CURRENT: 709/231

FIELD-OF-SEARCH: 709/223, 709/315, 709/100, 709/231, 709/230, 709/329, 713/1, 717/10, 717/5, 345/335

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected | Search ALL

	PAT-NO	SUE-DATE	PATENTEE-NAME	US-CL
☐	4791558	December 1988	Chaitin et al.	
☐	5386568	January 1995	Wold et al.	
☐	5524246	June 1996	Hurley et al.	
☐	5684967	November 1997	McKenna et al.	
☐	5754775	May 1998	Adamson et al.	370/261
☐	5794018	August 1998	Vrvilo et al.	709/231
☐	5812768	September 1998	Page et al.	709/217
☐	6047323	April 2000	Krause	709/201
☐	6067566	May 2000	Moline	707/500.1
☐	6141691	October 2000	Frink et al.	709/230
☐	6202096	March 2001	Williams et al.	709/227

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
WO 97/37303	October 1997	WO	
WO 99/15986	April 1999	WO	

OTHER PUBLICATIONS

Robertson, "Integrating Legacy Systems with Modern Corporate Applications," Communications of the ACM, 40, 39-46 (1997).
Gilbert, G., "Business Applications Cross the Border," Beyond the Enterprise, 2 pgs. (Oct. 1997).
Greenbaum, J., "Competitve Linking Update: CrossRoads to the Rescue," Hurwitz Balanced View Research Bulletin, 6 pgs. (Jun. 1997).
"Managing Emerging Telecommunications Technologies For Competitive Advantage," Versant Object Technolgy25 pgs. (1997).
"MQSeries Version 5--The Next Generation," 8 pgs. (Undated).
"ServiceGate.TM. Retail Service Manager Software," Bellcore, 6 pgs. (1997).
"Software Distributed Services Environment," Softwire, 4 pgs. (1997).
"Versant ODBMS Release 5," Versant Object Technologies, 8 pgs. (Oct. 1997).
Product Literature, "SAIC Project Profile," http:www.saic.com/telecom/profiles/ebss.html, 1 pg. (Oct. 1997).
Product Literature, "DSET Company Profile," http://www.dset.com/company/profile.html, 11 pgs. (Oct. 1997).
Product Literature, "Crossroads Software," http://www.crossroads-software.com/archsvcs.html, 5 pgs. (Jan. 1998),
Product Literature, "CrossRoads Customer Interaction,"http//www.crossroads-software.com/customer interaction.html, 1 pg. (Jan. 1998).
Product Literature, "CrossRoads," http://www.crossroads-software.com/processware.html, 2 pgs. (Jan. 1998).
Product Literature, "CrossRoads Partner," http://www.crossroads-software.com/partner information.html, 2 pgs. (Jan. 1998).
Product Literature, "Aberdeen Group," http://www.crossroads-software.com/aberdeenimpact.html, 3 pgs. (Jan. 1998).
Product Literature, "CrossRoads Management," http://www.crossroads-software.com/mgmt team.html, 3 pgs. (Jan. 1998).
Product Literature, "Enterprise Management Strategy," http://www.cai.com/products/unicent/whitepap.html, 23 pgs. (Feb. 1998).
Product Literature, "Computer Associates Press Release," http://www.cai.com/press/97dec/jasmine launch.html, 3 pgs. (Feb. 1998).
Product Literature, "Computer Associates Unicenter.RTM. TNG," http://www.cai.com/products/tng endorse.html, 6 pgs. (Feb. 1998).
Product Literature, "Computer Associates Product Description,"

http://www.cai.co[redacted]roducts/unicent/tng ov.html, 8[redacted]s. (Feb. 1998).
Product Literature, "OPAL Version 2," http://www.c[redacted].com/products/opal/wp1.html, 20 pgs. (Feb. 1998).
Product Literature, "Versant Object Technology," http://www.versant.com/whats new, 27 pgs. (Jan. 1998).
Product Literature, "SAIC The Vision," http://www.saic.com/telecom/index.html, 4 pgs. (Jan. 1998).
Product Literature, "Electronic Bonding," http://www.gteds.com/html/eb.html, 3 pgs. (May 1997).
Product Literature, "CLECs Use Internet Technology to Bond with Bells," http://www.zdnet.com/intweek/daily/97052e.html, 2 pgs. (Jan. 1998).
Product Literature, "MQSeries Business Partners Index," http://www.software.ibm.com/ts/mqseries/partners/partner.html, 1 pg. (Feb. 1998).
Product Literature, "MQSeries in General," http://www.software.ibmcom/ts/mqseries/library/brouchures/business, 3 pgs. (Feb. 1998).
Product Literature, "Massachusetts agencies exchange a wealth of data using MQSeries,"http://www.software.ibm.com/ts/mqseries/solutions/mass.html, 3 pgs. (Feb. 1998).
Product Literature, "Landmark Announces Performace Managment Solutin for MQSeries," http://www.software.ibm.com/news/2f46.html, 2 pgs. (Feb. 1998).
Products Literature, "Boole and Babage" http://www.boole.com/news/current/mqseries.html, 2 pgs. (Jan, 1998).
Product Literature, "Boole and Babage," http://www.boole.com/news/96%5Farchive/Commandmq.html, 2 pgs. (Jan. 1998).
Product Literature, "MSMQ: Interoperability," http://www.microsoft.com/ntserver/guide/msmq_interoperability.asp, 1 pg. (Jan. 1998).
Product Literature, "Microsoft Windows NT Server," http://www.microsoft.com./ntserver/community/msmqarchive.asp?A=7&B=5, 1 pg. (Jan. 1998).
Product Literature, "Microsoft Windows NT Server," http://www.microsoft.com/ntserver/guide/msmq--rev--patsey.asp?A=7 &B=5, 8 pgs. (Jan. 1998).
Product Literature, "Frontec--The AMTrix.TM. System Technical Overview," http://www.frontec.com/amtrixtechover.html, 5 pgs. (Jan. 1998).
Product Literature, "Frontec--The AMtrix.TM. System," http://www.frontec.com/amtrixsystem.html, 3 pgs. (Jan. 1998).
Product Literature, "Introduction to Messaging and Queuing," http://candy1, hursley.ibm.com:8001...r.cmd/book/HORAA101/, 31 pgs. (Jan. 1998).

ART-UNIT: 2151

PRIMARY-EXAMINER: Courtenay, III; St. John

ASSISTANT-EXAMINER: Nguyen; Van H.

ATTY-AGENT-FIRM: Schwegman, Lundberg, Woessner & Kluth, P.A.

ABSTRACT:

A system and method for exchanging data between two or more applications includes a data exchange engine and a number of adapters associated with a corresponding number of applications. Each of the adapters is customized to interface with a corresponding application and transforms data being transferred between the application and the data exchange engine. Data produced by a particular application is converted from a technology dependent form to a technology independent form by the corresponding adapter. In one embodiment, the format associated with a data stream is disassociated from the informational content of the data stream by the adapter. The informational content of the data stream is then transformed by the adapter into a common or generic format. The data exchange engine receives data in a technology independent form from each of its associated adapters and coordinates the routing of informational content to particular adapters associated with applications that have requested specific informational content. The adapters receiving the informational content from the data exchange engine transform the informational content having the common format into a data format compatible with, or specific to, their associated applications. A queuing mechanism is employed to construct a reliable asynchronous or pseudo-synchronous interface between disparate applications and systems. The data exchange engine may apply business rules or logic when

processing a requ[redacted] for particular informational [redacted]tent. User-specified routing logic may be appli[redacted] by the data exchange engine t[redacted]ispatch selected informational content to one or more destination applications.

56 Claims, 23 Drawing figures

L1: Entry 2 of 5 File: USPT Apr 4, 2000

US-PAT-NO: 6047390
DOCUMENT-IDENTIFIER: US 6047390 A

TITLE: Multiple context software analysis

DATE-ISSUED: April 4, 2000

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Butt; Farooq	Austin	TX		
Smith; Roger	Austin	TX		
Stewart; Katherine E.	Austin	TX		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Motorola, Inc.	Schaumburg	IL			02

APPL-NO: 08/ 995359 [PALM]
DATE FILED: December 22, 1997

PARENT-CASE:
This is based on U.S. patent application Ser. No. 08/703,261 filed Aug. 26, 1996, which is hereby incorporated by reference, and priority thereto for common subject matter is hereby claimed.

INT-CL: [07] G06 F 11/00

US-CL-ISSUED: 714/43; 395/675
US-CL-CURRENT: 714/43; 709/105

FIELD-OF-SEARCH: 714/43, 714/26, 714/38, 714/39, 714/41, 714/45, 714/47, 714/11, 714/12, 714/31, 714/37, 395/675, 395/677, 395/568, 395/569, 395/678, 395/680, 364/246, 364/550, 364/568

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

	PAT-NO	I[redacted]E-DATE	PATENTEE-NAME[redacted]	US-CL
☐	4462077	July 1984	York	364/300
☐	5129077	July 1992	Hillis	395/500
☐	5179702	January 1993	Spix et al.	395/650
☐	5212794	May 1993	Pettis et al.	395/700
☐	5333304	July 1994	Christensen et al.	395/575
☐	5390336	February 1995	Hillis	395/800
☐	5442758	August 1995	Slingwine	395/375
☐	5485574	January 1996	Bolosky et al.	395/183.11
☐	5490249	February 1996	Miller	395/183.14
☐	5506955	April 1996	Chen et al.	395/183.02
☐	5519867	May 1996	Moeller et al.	395/700
☐	5553235	September 1996	Chen et al.	395/182.18
☐	5560011	September 1996	Uyama	395/700
☐	5590056	December 1996	Barritz	364/550
☐	5594904	January 1997	Linnermark et al.	395/704
☐	5602729	February 1997	Krueger et al.	395/704
☐	5615333	March 1997	Juettner et al.	395/183.14
☐	5682328	October 1997	Roeber et al.	364/550
☐	5684945	November 1997	Chen et al.	395/182.18
☐	5704053	December 1997	Santhannam	284/383
☐	5748878	May 1998	Rees et al.	395/183.14
☐	5799143	August 1998	Butt et al.	395/183.14
☐	5847972	December 1998	Eick et al.	364/514A

OTHER PUBLICATIONS

Silicon Graphics, "Man Page Interface for IRIX 5.2", Release 5.2, 5 pgs.
Smith, et al. "Tracing with pixie", Division of Applied Sciences, Harvard University, Version 1.1, pp. 1-29.
Wulf, "SC'95 Table of Contents by Session", Supercomputing '95 Conference Dec. 3-8, Association for Computing Machinery, Inc., 9 pgs. (1995).
Chilimbi, et al., "StormWatch: A Tool for Visualizing Memory System Protocols", AT&T Bell Laboratories, Sections 1-6, 16 pgs.
IBM Corp., "Program Visualizer (PV) Tutorial and Reference Manual", Release 0.8.1, 142 pgs.
Eggers, et al., "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor", Proceedings of the 1990 ACM Sigmetrics Conference, pp. 37-47 (1990).
Kimelman, et al., "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior", Visualization '94 Conference, IBM Thomas J. Watson Research Center, pp. 1-14 (1994).
Stephens, et al., "Instruction Level Profiling and Evaluation of the IBM RS/6000", ACM (1991).

ART-UNIT: 275

PRIMARY-EXAMINER: Beausoliel, Jr.; Robert W.

ASSISTANT-EXAMINER: Iqbal; Nadeem

ATTY-AGENT-FIRM: H[redacted]1; Daniel D.

ABSTRACT:

A method for multiple context analysis of software applications in a multiprocessing (22, 23), multithreaded computer environment utilizes instrumentation code inserted (54, 55) into the applications. For each execution (67) of the application (60), a context set is selected (62). Execution of the instrumented code (67) provides information for analysis in an instrumentation buffer (82) addressed by a reserved register (80) or buffer pointer. The operating system is responsible for providing in the reserved register (80) the address of the instrumentation buffer (82) appropriate for each instrumented context executed. When the application (60) is done with an instrumentation buffer (82), the buffer may be processed by filter software (68). The combination of using a reserved register (80) and allowing the operating system to keep track of relevant context switches allows applications to be instrumented (54, 55) for various context sets without the necessity of modifying (53) or recompiling (52) the application software (60).

2 Claims, 11 Drawing figures

Generate Collection | Print

L1: Entry 3 of 5 File: USPT Aug 25, 1998

US-PAT-NO: 5799143
DOCUMENT-IDENTIFIER: US 5799143 A

TITLE: Multiple context software analysis

DATE-ISSUED: August 25, 1998

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Butt; Farooq	Austin	TX		
Smith; Roger	Austin	TX		
Stewart; Katherine E.	Austin	TX		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Motorola, Inc.	Schaumurg	IL			02

APPL-NO: 08/ 703261 [PALM]
DATE FILED: August 26, 1996

INT-CL: [06] G06 F 11/00

US-CL-ISSUED: 395/183.14; 395/678, 395/568
US-CL-CURRENT: 714/38; 709/108, 712/227

FIELD-OF-SEARCH: 395/183.14, 395/183.21, 395/183.13, 395/183.01, 395/184.01, 395/183.09, 395/675, 395/677, 395/678, 395/680, 395/568, 395/569, 395/500, 395/704

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected | Search ALL

	PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
☐	4462077	July 1984	York	395/183.21
☐	5129077	July 1992	Hillis	395/500
☐	5179702	January 1993	Spix et al.	395/650
☐	5212794	May 1993	Pettis et al.	395/183.14
☐	5333304	July 1994	Christensen et al.	395/183.01
☐	5390336	February 1995	Hillis	395/800
☐	5442758	August 1995	Slingwine et al.	395/474
☐	5485574	January 1996	Bolosky et al.	395/184.01
☐	5490249	February 1996	Miller	395/183.14
☐	5519857	May 1996	Moeller et al.	395/181
☐	5553235	September 1996	Chen et al.	395/182.18
☐	5590056	December 1996	Barritz	395/183.14
☐	5594904	January 1997	Linnermark et al.	395/183.11
☐	5602729	February 1997	Krueger et al.	395/184.01

OTHER PUBLICATIONS

"Man Page Interface for IRIX 5.2", pub. by Silicon Graphics, Release 5.2, 5 pgs.
William A. Wulf, "SC'95 Table of Contents by Session", pub.by Assoc. for Computing Machinery,Inc., for Supercomputing Conf. Dec.3-5, 1995, 9 pgs.
Michael D.Smith, "Tracing with pixie", Version 1.1, pp. 1-29.
Chilimbi, et al, "StormWatch: A Tool for Visualizing Memory System Protocols", pub. by AT&T Bell Labs, for SC'95 TOCS, pp. 1-16.
Kimelman, et al, "Strata-Various:--Multi-Layer Visualization of Dynamics in Software System Behavior", pub. by IBM Thomas J.Watson Res.Center Jul. 26, 1994, for Visualization '94, pp. 1-14.
Chriss Stephens, et al., "Instruction Level Profiling and Evaluation of the IBM RS/6000", pub. by ACM 1991, pp. 180-189.
Susan J.Eggers et al, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor", pub. by ACM, 1990, pp. 27-37.
Michelle Harr, "Program Visualizer (PV) Tutorial and Reference Manual RElease 0.8.1", pub.by IBM Software Solutions, pp. 1-142.

ART-UNIT: 275

PRIMARY-EXAMINER: Palys; Joseph

ATTY-AGENT-FIRM: Hayden; Bruce E. Hill; Daniel D.

ABSTRACT:

A method for multiple context analysis of software applications in a multiprocessing (22, 23), multithreaded computer environment utilizes instrumentation code inserted (54, 55) into the applications. For each execution (67) of the application (60), a context set is selected (62). Execution of the instrumented code (67) provides information for analysis in an instrumentation buffer (82) addressed by a reserved register (80) or buffer pointer. The operating system is responsible for providing in the reserved register (80) the address of the instrumentation buffer (82) appropriate for each instrumented context executed. When the application (60) is done with an instrumentation buffer (82), the buffer may be processed by filter software (68). The combination of using a reserved register (80) and allowing the operating system to keep track of relevant context switches allows applications to be instrumented (54, 55) for various context sets without the necessity of modifying (53) or recompiling (52) the application software (60).

22 Claims, 11 Drawing figures

L1: Entry 4 of 5 File: USPT May 1, 1990

US-PAT-NO: 4922414
DOCUMENT-IDENTIFIER: US 4922414 A

TITLE: Symbolic language data processing system

DATE-ISSUED: May 1, 1990

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Holloway; John T.	Belmont	MA		
Moon; David A.	Cambridge	MA		
Cannon; Howard I.	Lexington	MA		
Knight; Thomas F.	Belmont	MA		
Edwards; Bruce E.	Belmont	MA		
Weinreb; Daniel L.	Somerville	MA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Symbolics Inc.	Cambridge	MA			02

APPL-NO: 07/ 078724 [PALM]
DATE FILED: September 8, 1987

PARENT-CASE:
This is a division of application Ser. No. 450,600, filed Dec. 17, 1982, now abandoned.

INT-CL: [05] G06F 9/00

US-CL-ISSUED: 364/200; 364/255.1, 364/255.2, 364/255.3, 364/255.4, 364/255.5, 364/255.7, 364/255.8
US-CL-CURRENT: 711/207; 711/216

FIELD-OF-SEARCH: 364/2MSFile, 364/9MSFile

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected | Search ALL

	PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
☐	4587610	March 1986	Rodman	364/200
☐	4680700	July 1987	Hester et al.	364/200
☐	4682281	July 1987	Woffinden et al.	364/200
☐	4730249	March 1988	O'Quin II, et al.	364/200

ART-UNIT: 237

PRIMARY-EXAMINER: Shaw; Gareth D.

ASSISTANT-EXAMINER: Mills; John G.

ATTY-AGENT-FIRM: Sprung Horn Kramer & Woods

ABSTRACT:

A symbolic language data processing system comprises a sequencer unit, a data path unit, a memory control unit, a front-end processor, an I/O and a main memory connected on a common Lbus to which other peripherals and data units can be connected for intercommunication. The system architecture includes a novel bus network, a synergistic combination of the Lbus, microtasking, centralized error correction circuitry and a synchronous pipelined memory including processor mediated direct memory access, stack cache windows with two segment addressing, a page hash table and page hash table cache, garbage collection and pointer control, a close connection of the macrocode and microcode which enables one to take interrupts in and out of the macrocode instruction sequences, parallel data type checking with tagged architecture, procedure call and microcode support, a generic bus and a unique instruction set to support symbolic language processing.

6 Claims, 22 Drawing figures

End of Result Set

Generate Collection | Print

L1: Entry 5 of 5 File: USPT Dec 12, 1989

US-PAT-NO: 4887235
DOCUMENT-IDENTIFIER: US 4887235 A

TITLE: Symbolic language data processing system

DATE-ISSUED: December 12, 1989

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Holloway; John T.	Belmont	MA		
Moon; David A.	Cambridge	MA		
Cannon; Howard I.	Sudbury	MA		
Knight; Thomas F.	Belmont	MA		
Edwards; Bruce E.	Belmont	MA		
Weinreb; Daniel L.	Arlington	MA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Symbolics, Inc.	Cambridge	MA			02

APPL-NO: 07/ 129921 [PALM]
DATE FILED: December 3, 1987

PARENT-CASE:
This application is a continuation of application Ser. No. 450,600, filed 12/17/82, now abandoned.

INT-CL: [04] G06F 9/00

US-CL-ISSUED: 364/900; 364/951.5, 364/970.4, 364/972.1, 364/967.4
US-CL-CURRENT: 711/216; 707/206

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected | Search ALL

	PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
☐	3670307	June 1972	Arnold et al.	
☐	3670309	June 1972	Amdahl et al.	
☐	4128882	December 1978	Dennis	364/900
☐	4130885	December 1978	Dennis	364/900
☐	4447875	May 1984	Bolton et al.	364/200

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
42540	December 1978	AT	
91-5310	October 1972	CA	
1017871	August 1974	CA	
995823	August 1976	CA	
1023056	December 1977	CA	

ART-UNIT: 237

PRIMARY-EXAMINER: Shaw; Gareth D.

ASSISTANT-EXAMINER: Mills; John G.

ATTY-AGENT-FIRM: Sprung Horn Kramer & Woods

ABSTRACT:

A symbolic language data processing system comprises a sequencer unit, a data path unit, a memory control unit, a front-end processor, an I/O and a main memory connected on a common Lbus to which other peripherals and data units can be connected for intercommunication. The system architecture includes a novel bus network, a synergistic combination of the Lbus, microtasking, centralized error correction circuitry and a synchronous pipelined memory including processor mediated direct memory access, stack cache windows with two segment addressing, a page hash table and page hash table cache, garbage collection and pointer control a close connection of the macrocode and microcode which enables one to take interrupts in and out of the macrocode instruction sequences, parallel data type checking with tagged architecture, procedure call and microcode support, a generic bus and a unique insruction set to support symbolic language processing.

6 Claims, 23 Drawing figures

End of Result Set

Generate Collection | Print

L10: Entry 1 of 1 File: USPT Sep 17, 2002

DOCUMENT-IDENTIFIER: US 6453356 B1
TITLE: Data exchange system and method

Brief Summary Text (8):
Fourth and fifth generation languages, termed 4GL and 5GL, would appear to offer a partial solution to the data interchange problem. These and other similar languages, such as those used to construct "business objects," are optimized around the construction of applications and user interfaces for the primary purpose of providing powerful querying and reporting tools. The objects created by such languages typically define access paths to data residing in databases. The objects can be combined in various ways to create complex queries and for building powerful report generators. Fourth and fifth generation languages are not well suited for transporting vast amounts of data from one application to one or more other applications in a reliable and efficient manner. Although business objects constructed using 4GL and 5GL techniques do carry with them a certain amount of data, this data is typically data resulting from a query that is transported with an object definition for the purpose of performing additional analysis on the data.

Drawing Description Text (15):
FIG. 18 is a class structure diagram showing public and non-public interfaces associated with various file based and database based queuing processes;

Detailed Description Text (51):
The Common Base Class is an abstract base from which the Common Object Class is derived. An inheritance tree graphically depicting the Common Base Class is shown in FIG. 17. The main purpose of this class is to provide a single object naming and typing mechanism to aid in object tree searches and traversals. The Common Base Class is characterized in the following code-level example.

Detailed Description Text (61):
The following Common Object retrieval methods are used internally by the GetAttributeValue( ) and SetAttributeValue( ) methods to search the attribute list of a Common Object and to locate a specific Common Attribute instance. These retrieval methods may be used by application developers as well. Each of these methods require a fully dot(.) delimited Distinguished Name and will recursively walk all relative levels of nesting to retrieve the relevant object/attribute.

Detailed Description Text (134):
An error/event may occur at a very low level in the code (e.g., database space exhausted). It is important to report this low level event, but it is also important to report the context of what was trying to be achieved within the application when this low level error occurred. The application developer is provided with macros to define a context within the developer's code. The set of macros provided for this purpose include: INIT_CONTEXT; CONTEXT_BEGIN; and CONTEXT_END. In general, every function using the context macros should first use the macro INIT_CONTEXT. It is noted that, if INIT_CONTEXT is not called before defining CONTEXT_BEGIN, the code may not compile.

Detailed Description Text (135):
The beginning of a context may be defined using the macro CONTEXT_BEGIN, and the end of a context can be defined using the macro CONTEXT_END, as is indicated in the following example. The CONTEXT_BEGIN macro takes the argument Context Number. This context number is used to access the Context Catalog of an application and to retrieve the context string. It is noted that nested contexts are generally not

allowed. If a CONTEXT_BEGIN is called before the previous context is ended, an implicit CONTEXT_END for the previous context is assumed. The following example is provided:

Detailed Description Text (142):
The macro CONTEXT_BEGIN, described in the following example, checks whether dx_init_context is initialized or not. The significance of this check is to make sure that the function does not compile if INIT_CONTEXT is not called before the first occurrence of CONTEXT_BEGIN. It then initializes the DX_ContextObject pointer to point to a new DX_ContextObject instance storing the context string specified by the context number argument.

Detailed Description Text (144):
The macro CONTEXT_END deletes the DX_ContextObject instance created by CONTEXT_BEGIN, as can be seen in the following example.

Detailed Description Text (147):
The following statistics are typically recorded when monitoring is performed on the queue: (1) number of the messages processed in the system input queue; (2) the average message cache time in the system input queue, by priority; and (3) the average message processing time from the system input queue, by priority. The following statistics are typically recorded when monitoring is performed on the disk space and database table space usage: (1) the percentage of the disk space usage if the queue storage type is FLATFILE; and (2) the percentage of the table space usage if the queue storage type is DATABASE.

Detailed Description Text (151):
The monitor writes the log file in the same format irrespective of whether the queue is file based or database based. The format of the report file is provide in Table 1 below:

Detailed Description Text (153):
When disk space and/or database table space usage is being monitored (i.e., SYSTEM_INFO_MONITOR=ON), the monitor writes the usage of the disk where the queue files located into an ASCII file on a regular time interval if the queue is file based. The name of the file follows the system file name schema (e.g., _SysInfo.Mon_Log). The maximum file size is defined in the system configuration file. After the file reaches its maximum size, the monitor moves it to a backup copy named _SysInfo.Mon_Log.bk and creates and writes the performance into the new _SysInfo.Mon_Log file. The system retains only one backup copy of the monitor log files. The format of the report file is give below in Table 2:

Detailed Description Text (154):
where, Time represents the time stamp of the record in the log file; and DiskUsage represents the percentage of the file disk usage if STORAGE_TYPE=FLATFILE. Again, the data fields are delimited by a comma. It is noted that the database table space usage is generally not available to the application user, such that only the system administrator has the privilege to view it. As such, the monitor does not perform monitoring for the database table space usage.

Detailed Description Text (157):
The Monitor Object is instantiated by the DX_SysConfigObject instance or by calling the static method DX_Monitor::Instance( ) directly. There is only one DX_Monitor thread running per executable component. The monitor thread is spawned whenever the MONITOR in the system configuration file is triggered to ON. The monitor thread exists until the MONITOR is triggered to OFF. The implementation of the monitor impacts three areas. The Queue Manager provides the queue performance data. The DX_SysConfigObject provides the configuration change handling and the monitor object instantiation. The DX_Monitor Object spawns or kills the monitor and generates the monitor log files or log table in the database. The methods added in the DX_Queue classes are listed below:

Detailed Description Text (177):
Concerning data exchange system security, the basic security control is focused on the queue files access or the database tables access. The file access control requires the application user to be in a specific user group. The user group should be set before the application runs. The database table access control requires the application users to have the correct user name and user password. The user name and user password may be set in the environment variables or be hard coded in the

application progr[redacted]. In one embodiment, all appli[redacted]ions share one database user account. This use[redacted] account has privileges to creat[redacted]pdate/delete tables in the view.

Detailed Description Text (187):
A skeleton main( ) function is provided to illustrate the system initialization and startup procedures required for each component of the data exchange system application. This includes operations such as database connection, system resource configuration, thread control, etc. In addition, a System Health Monitor thread is provided which, on a timed interval, polls all system resources to ensure that system operation can continue. This thread invokes the system checking operations System Configuration Object. The sample code provided below illustrates the ease of initializing system components and application operations. This sample is part of the DX_Engine executable, which serves as the core of the data exchange system.

Detailed Description Text (196):
A further description of a data exchange system queuing methodology in accordance with one embodiment of the present invention will now be described. In order to provide a clean "buffered API," a queued request approach is used. Use of interface queues allows the caller of the API to send its request irrespective of whether the engine core and another outgoing adapter are running. The queue interface approach of the instant embodiment also provides a mechanism for buffering the load that may be placed upon a server from multiple clients. It also provides the ability to scale the number of database servers that can process any given queue in parallel.

Detailed Description Text (197):
As was discussed previously, two types of priority based queues are used, namely, the incoming Receive Queues and the outgoing Send Queues. Each outgoing adapter will have its own outgoing queue so that any interface specific translation or routing may be performed outside the engine core. Each instance of the DX_Engine executable has one or more input queues, although only one is allowed for file-based queues, and one or more output queues. An instance of the DX_QueueManager class is used as a central proxy to all queue access, and will be mutex protected and record-lock protected, for file-based implementation, or row lock protected, for database implementations, to prevent data contention.

Detailed Description Text (198):
Two types of queues are provided, file and database queues, both of which are fairly simple implementations that allow for a clean breakup to the API. Priority based queuing is provided so that requests of high importance can be serviced quickly. Multiple queues are used to provide this level of functionality, but the implementation is logically transparent to users. A user perceives that there is only one logic queue with objects of different priority in it.

Detailed Description Text (199):
The file storage or database tables for the queue are created at running time and deleted by queue administration process. There are four types of pre-defined priority: NONURGENT; NORMAL; URGENT; and INTERACTIVE in order of increasing priority. INTERACTIVE is the highest priority, which can block any request having other priorities. The priority algorithm ensures that the Dequeue operation always returns successfully when the queue is not empty, prevents starvation of lower priority entries, and ensures more frequent visits on higher priority queues. The priority algorithm is implemented on a weighted basis of each priority.

Detailed Description Text (200):
Support for parallel gateways is only available to a database queue implementation. File-based queue implementations will not guarantee single delivery, i.e., one object might be dequeued by multiple process at the same time. All parallel access should be completely transparent to any participating gateway. The only areas of common resources between any parallel gateways are the Enqueue and Dequeue operations. The design of the Enqueue/Dequeue module ensures that parallel access is made possible without any deadlocks or duplicated queue entries by using the database supplied row-level locking.

Detailed Description Text (201):
Since the external API is limited to the Enqueue/Dequeue API, the only limit to multiple access is the row-level table locking that the database supports. The file based queue mechanism uses simple file record-lock to protect from multiple updates to the file from multiple threads. The queue access operations for file-based

implementation are thread-safe, such that all the operations are mutex protected.

Detailed Description Text (202):
The Queue Manager public interface makes use of the DX_QueueTransaction object for transaction control. The Enqueue( ), Dequeue( ), Commit( ), and Rollback( ) methods take pass-in argument of an instance of the DX_QueueTransaction class which belongs to a running thread. The transaction object contains an ordered list of operations performed in this transaction. For file-based implementations, all operations are maintained in buffered memory and are not written into file storage until commit time. For database implementations, the database provided rollback mechanism is employed, with each transaction using its own unique run-time database context.

Detailed Description Text (221):
Dequeue( ) attempts to find a queue object marked as NORMAL_OBJECT first from the memory buffer. If it can not find one, it will try to find one from the queue files. If Dequeue( ) finds a queue object marked as NORMAL_OBJECT in a file, it creates a queue object, marks it as ACTIVE_OBJECT, inserts it into the memory buffer, and de-serializes the Common Object it refers to and returns the Common Object to the caller. During this process, if Dequeue( ) can not de-serialize the Common Object, it will mark the queue object as DELETED_OBJECT in the queue file and continue its search.

Detailed Description Text (223):
DX_DBQueue is the database-based implementation of the DX_Queue interface. Its instance is mapped to a single table per queue at run-time. The order of the records is determined by the enqueue time stamp. All dequeue operations are sorted by priority and enqueue time stamp. An illustrative example of a DX_DBQueue implementation is provided as follows:

Detailed Description Text (227):
The invocation of the Enqueue( ) and Dequeue( ) API is effected by sending a request to the Queue Manager Object. In response to a request, the Queue Manager Object locates the correct queue and populates the operations to that queue object. When Enqueue( ) is invoked, the object is placed into a buffer list and will not be collapsed into a data stream until Commit( ) is invoked. Until the commit time, the DX_IndexObject attribute of the DX_QueueOperation object retains the real meaning, which may be used in connection with a rollback operation if the commit operation is not successfully executed. For purposes of serialization, if the queue is a database, the row-level locking provided by the database is used. If the queue is a file, file access control is used. When a Common Object is serialized at commit time, a priority tag is appended to its private attribute list, such that when the Common Object is dequeued, its priority can be easily determined.

Detailed Description Text (228):
When Dequeue( ) is invoked, the oldest entry with the proper priority is retrieved with the marshalled object then being instantiated as a Common Object using the Demarshal( ) method invocation. This mechanism provides database row-locking on read events to prevent parallel gateways reading the same queued requests. If the queue is file based, record lock is used. The logic to determine which priority queue should be invoked on implemented inside the DX_Queue object.

Detailed Description Text (232):
Objects are stored and retrieved from a persistent store in an efficient manner. Two types of object storage, termed relational database storage and file storage, are implemented for this purpose. The object persistency is implemented using Marshal( ), Demarshal( ), and DeleteStorage( ) methods of the DX_CommonObject class, where a parameter is passed to indicate storage type. A policy for object caching and sharing may also be employed.

Detailed Description Text (233):
Since database implementation may vary among venders and file based persistency is needed, the object persistency model has been developed to be independent from any database storage technology. This approach involves collapsing the object into a raw binary data stream, and uses various headers to denote any marker and priority related information. The headers of a Common Object typically include OID and priority tag information. Table 4 provide below illustrates a how a Common Object may be stored in a typical database.

Detailed Description Text (234):

As stated previous[redacted] all database access is hidden [redacted]side the Enqueue( ) and Dequeue( ) methods of DX_QueueManager and Marshal( ) [redacted] Demarshal( ), and DeleteStorage( ) methods of DX_CommonObject. In one embodiment, all vendor specific access mechanisms may be delivered in a separate set of database enabled libraries.

Detailed Description Text (238):
File-based database access is invoked using the Marshal( ), Demarshal( ), and DeleteStorage( ) methods from the Common Object with the output argument set to a file. Each object may be stored to a separate file where the filename incorporates the object ID. By using the object counter mechanism as a directory name, files can be spread evenly across a file system for better access time.

Detailed Description Text (240):
A transaction object should be created and destroyed as a local object of a thread. If neither Commit( ) nor Rollback( ) was called when a thread exits, all operations executed by this thread are rolled back. For a database implementation, the native transaction control mechanism of the database is used. For a file implementation, a transaction object contains a list recording of all operations in the current transaction. This memory buffer is used to buffer all queue operations and will not be written into file storage until commit time. Rollback( ) removes the operations from that memory buffer for operations that have not been committed. Since Commit( ) might fail, Rollback( ) also cleans the entries in the queue files and Common Object file for those operations that failed at commit time.

Detailed Description Paragraph Table (2):
enum EclassTypes { DX_OBJECT = 0, DX_ATTRIBUTE, DX_LIST, DX_MULTIVALUE, DX_INTEGER, DX_STRING, DX_STRINGVAL, DX_DATE, DX_TIME, DX_REAL, DX_BLOB, DX_BLOBVALUE, DX_VERSION, DX_OID_CLASS }; enum EreturnCodes { NOT_FOUND = 0, SUCCESS, FAILED, TIME_OUT, ACCESS_DENIED, DUPLICATE_OBJECT, NO_OBJECT, NO_ATTRIBUTE, INVALID_ATTRVAL, INVALID_ARGS, INVALID_OPERATION, SYSTEM_ERROR }; enum EstorageTypes { FLATFILE = 0, DATABASE }; enum ElistType { PUBLIC = 0, PRIVATE }; static const int MAX_NAME_LEN=255; static const int MAX_DOTTED_NAMELEN=8096; static const int MAX_MULTI_VAL=255; static const int MAX_BLOB_SIZE=2048; static const int OID_LEN=128; static const int MAX_FILE_NAME=256; static const int MAX_PID_LEN=12; static const int MAX_LINE_SIZE=100;

Detailed Description Paragraph Table (3):
class: DX_CommonObject: public DX_CommonBase { RWDECLARE_COLLECTABLE(DX_CommonObject); friend class DX_ListObject; friend class DX_FileSubQueue; public: virtual.about.DX_CommonObject(); // Constructors /*****/ //If name==NULL, the name is set to "NOT_SET". //If name is ""or contains a "." it will be set to "INVALID_NAME" /*****/ DX_CommonObject(const char* name=0); /*****/ //Create a copy of an entire DX_CommonObject, //but with it's own unique OID value /*****/ DX_CommonObject* Clone(); /*****/ //All AddAttribute and AddPvtAttribute methods take ownership of //the pointers passed in to them. Do NOT delete the pointers after //a call to AddAttribute and AddPvtAttribute. The pointers will be //deleted when the container DX_CommonObject or DX_ListObject is deleted. // //NOTE: When using the following two methods for creating a DX_STRING attribute // // AddAttribute(const char* name, const int type, const void* value) // and AddPvtAttribute(const char* name, const int type, const void* value) // // the value is defaulted to be of type const char* and not UNICHAR* // Misuse will lead to unexpected behavior. /*****/ EreturnCodes AddAttribute(DX_CommonAttribute* newAttr); EreturnCodes AddAttribute(DX_ListObject* newAttr); EreturnCodes AddAttribute(DX_CommonObject* newAttr); EreturnCodes AddAttribute(const char* name, const int type, const void* value); EreturnCodes AddPvtAttribute(DX_CommonAttribute* newAttr); EreturnCodes AddPvtAttribute(DX_ListObject* newAttr); EreturnCodes AddPvtAttribute(DX_CommonObject* newAttr); EreturnCodes AddPvtAttribute(const char* name, const int type, const void* value); /*****/ //DeleteAttribute and DeletePvtAttribute will remove the named attribute //from the container DX_CommonObject or DX_ListObject and delete the named //attribute's pointer /*****/ EreturnCodes DeleteAttribute(const char* name); EreturnCodes

DeletePvtAttribute(const char* name);
/**/
//RemoveAttribute will remove the named attribute from the container
//DX_CommonObject or DX_ListObject but will not delete the named //attribute's
pointer /**/
EreturnCodes RemoveAttribute(const char* name); EreturnCodes
RemovePvtAttribute(const char* name);
/**/ //Do NOT delete
the pointer that is returned to you, it still is owned by //the container
DX_CommonObject or DX_ListObject. You CAN use the any of //the attribute class
methods for the pointer.
/**/ void*
GetAttribute(const char* name); void* GetPvtAttribute(const char* name); void
PrintContents(); //The caller of Demarshal is responsible for object's memory
allocation static DX_CommonObject* Demarshal(char* ObjOid, int type, int
ContextIndex); EreturnCodes Marshal(int type, int ContextIndex); static EreturnCodes
DeleteStorage(const char* oidVal, int type, int ContextIndex); //The caller of
GetOID is responsible for deleting the memory of the returned char* char* GetOID();
EPriorityCode GetPriority(); protected: void* Find(const char* name); static
EreturnCodes RestorePersistentObject(const char* oidVal); static EreturnCodes
DeletePersistentObject(const char* oidVal, int type); private: //Data Members
DX_ListObject* PrivateList; DX_ListObject* PublicList; static void
Delete(RWDlistCollectables* list); static int Copy(RWDlistCollectables* dest, const
RWDlistCollectables *src); static EreturnCodes CopyPersistentObject(char* filename);
static EreturnCodes CopyFile(char* filename,char* copyfilename); void
saveGuts(RWFile& file) const; void saveGuts(RWvostream& stream) const; void
restoreGuts(RWFile& file); void restoreGuts(RWvistream& stream); const int check()
const; DX_Boolean updateListOids(DX_ListObject* Plist, DX_OID *Poid); DX_Boolean
updateObjectOids(DX_CommonObject* Pobj, DX_OID *Poid); DX_Boolean
ValidateNaming(const char* name); #ifdefDX_DATABASE EreturnCodes
InsertIntoTable(char* filename, int ContextIndex); static EreturnCodes
RetrieveFromTable(char* filename, char* oid, int ContextIndex); EreturnCodes
UpdateTable(char* filename, int ContextIndex); #endif //use to cal. the number of
bytes needed to store an object using RWFile RWspace binaryStoreSize() const;}; };

Detailed Description Paragraph Table (42):
enum EstorageTypes { FLATFILE = 0, DATABASE }; enum EQueueOperation { ENQUEUE = 0,
DEQUEUE }; static const int NUM_PRIORITIES = 4; enum EPriorityCode { NONURGENT = 0,
NORMAL, URGENT, INTERACTIVE };

First Hit Fwd Refs

Generate Collection Print

L3: Entry 1 of 4 File: USPT Sep 17, 2002

DOCUMENT-IDENTIFIER: US 6453356 B1
TITLE: Data exchange system and method

Brief Summary Text (15):
Process monitoring, tracing, and logging are provided to track the progress of data passing through the data exchange engine and to detect and correct processing errors. In the case of a processing anomaly, the data exchange engine effects a rollback of failed transactions to preclude the loss of data. Performance statistics may also be provided.

Detailed Description Text (39):
Also shown in FIG. 9 is a statistics monitor module 264 and an associated statistics log 276 which are used to provide monitoring and tracking of data as it moves through the data exchange system. The statistics monitor module 264 also provides historical performance information on queues and historical information on system resource usage. As will be described in greater detail hereinbelow, the statistics monitor module 264 provides a means for logging and tracing a given application. Logging reveals the state of the application at the time of an error, while tracing provides a description of all software events as they occur. The tracing information may be used for tracking the application, state, and other related operations. The tracing information may be used in conjunction with the logging information to determine the cause for an error since it provides information about the sequence of events prior to an error.

Detailed Description Text (109):
Having discussed in detail various aspects of the Common Object design hereinabove, a more comprehensive description of a data exchange system infrastructure in accordance with one embodiment of the present invention is provided below. The various aspects of the system infrastructure described in greater detail hereinbelow include: configuration management; logging and tracing; context definition; performance and statistics monitoring; administration and maintenance; security; processing thread pool; internationalization; and process procedures.

Detailed Description Text (111):
Parameters of a component, such as trace/log settings, may be changed at run time. For this purpose, configuration management tools provide a command line interface and a Web interface for viewing and configuring various parameters at run time. It is noted that various components of the data exchange system may be running on different machines. Thus, the configuration management utility provides the ability to view and modify the parameters of a component running on a different machine, possibly on a different platform. The Web interface of the configuration management utility provides the requisite connectivity to a remote machine and provides the capability for performing remote configurations.

Detailed Description Text (112):
When initiated, a component creates an instance of a System Configuration Object (DX_SysConfigObject) that stores the current parameter settings. The component also registers for a Signal/Event so that it is informed of changes to the configuration using the dynamic configuration command line interface/web interface. When a user

wants to change the run time parameters of a component (identified by the process ID and the machine on which it is running), a signal/event is sent to the component to update its configuration. A signal/event handler invokes the ReconfigParameters ( ) method on the DX_SysConfigObject, which takes care of reconfiguring the various controller objects, such as DX_TraceLogObject, DX_QueueManager, and DX_ThreadController for example. The System Configuration object, DX_SysConfigObject, is a singleton object that initializes and controls the configuration of a component in the data exchange system, such as logging/tracing levels, thread controller, queuing, and performance monitoring. A singleton object, as understood in the art, refers to a C++ nomenclature meaning that only a single instance of the class may exist within a single executable. A singleton object is most commonly used when controlling system wide resources.

Detailed Description Text (118):
The logging and tracing utility provides for logging and tracing during execution of a component. As discussed previously, logging reveals the state of the component at the time of an error, while tracing provides a description of all software events as they occur. The tracing information may be used for tracking the component-state, and other related operations. It may be used in conjunction with the logging information to determine the cause of an error, as it provides information about the sequence of events prior to an error.

Detailed Description Text (119):
A component that intends to record a log/trace message indicates the category to which the message belongs. The log and trace messages are recorded in two different files whose names are derived from the name of the application, as indicated in the following example: (<componentName><pid>.log and <componentName><pid>.trc, respectively)

Detailed Description Text (121):
Tracing involves recording three types of messages, which are typically specified by the developer. The INFORMATION (SYS_INFO and APP_INFO) message provides a record of specific events that occur during the execution of the component, such as beginning of a new control thread. SYS_INFO is used within the DX libraries and APP_INFO is to be used by applications using the DX libraries. The TRACE (SYS_TRACE and APP_TRACE) message provides a detailed record of various software events that occur during the course of normal execution of the component. SYS_TRACE is used within the DX libraries and APP_TRACE is to be used by the applications using the DX libraries. The AUDIT message provides a record of all the information sent or received by various components of the data exchange system.

Detailed Description Text (122):
A configuration file is used to store trace/log related parameters in one or more directories. The_TRACE_LOG_DIR directory is used to store the trace/log files. If this parameter is not specified, it is set by default to the current working directory. The tracing level associated with the SYS_INFO, APP_INFO, SYS_TRACE, APP_TRACE, and AUDIT parameters may be specified as ON or OFF. The default value for any trace level is OFF. The WARNING, MAJOR, and CRITICAL parameters may also be specified as ON or OFF. It is noted that there exists a hierarchical relationship between these three error levels. For example, if WARNING is ON, it implies that all the error levels are ON. If MAJOR is ON, then CRITICAL is ON.

Detailed Description Text (123):
The TRACE_LOG_SIZE parameter controls the maximum size of a trace/log file. When the trace or log file reaches the specified size, it is moved into the files named <componentname><pid>.trc.bk or <componentname><pid>.log.bk, respectively. The default value for the trace/log file size is 100K bytes.

Detailed Description Text (124):
A trace statement is typically used to write a developer specified string to the

<ComponentName><pid>.trc file if the trace category specified by the developer, which is generally hard coded in the program, is ON at run time. A log statement is generally used to write a specified error message to <ComponentName><pid>.log file if the category specified by the developer, which is generally hard coded in the program, is set to ON at run time. It is noted that the developer typically specifies an error number that is used to retrieve the error message from an external Message Catalog.

Detailed Description Text (125):
In order to write a message into the log/trace file, the developer may use the macro DX_TL as shown below: DX_TL(DX_ARGS,Category, StringToLog/ErrorNumber[,arg 1 [,arg2]]);

Detailed Description Text (126):
The macro DX_ARGS includes parameters such as filename, line number, time and thread ID that are automatically written into the trace/log messages. Category is specified by the following enumerated data types:

Detailed Description Text (128):
StringToLog is specified by the developer as a trace message and is written into the <ComponentName><pid>.trc file (type char *). For Example, an AUDIT message could include the details of a Common Object typically stored as a formatted string. ErrorNumber is specified by the developer for a log message (e.g., when the category is CRITICAL, MAJOR or WARNING), and is used to index into a Message Catalog to retrieve the error message to be logged. The error message numbers are defined as an enumerated type as shown below:

Detailed Description Text (130):
A component using the Tracing/Logging Utility needs to include the following statement: #include "traceLogUtil.h". The definition of DX_TraceLogObject and other related definitions are provided as follows:

Detailed Description Text (132):
The Write( ) method in the DX_TraceLogObject calls WriteFormattedTraceLog( ) of DX_TraceLogFormatControl Object to write to the trace/log stream in a specific format. Thus, the implementation of the DX_TraceLogFormatControl object may be changed to accommodate the needs of users who would want to implement a desired formatting style. The contents of Trace/Log messages that are logged include the following: Category; File name; Line Number; Thread ID; Time; and Context.

Detailed Description Text (133):
The System Configuration Object contains an instance of the Trace/Log Object, which is initialized with parameter values specified in the Configuration file. When the user modifies the Trace/Log parameters at run time, typically by use of the DX_ConfigSet command, a signal is sent to the applicable component which calls the method ReconfigParameters( ) on the System Configuration Object to re-initialize the parameters. This method, in turn calls ReconfigParameters( ) on the DX_TraceLogObject.

Detailed Description Text (139):
The method GetContextStr( ) is called by every log/trace statement to include the current context information in the message. GetContextFromCatalog( ) is used to retrieve the Context information from the context catalog.

Detailed Description Paragraph Table (22):
enum DX_TL_CATEGORY { CRITICAL, MAJOR, WARNING, SYS_INFO APP_INFO SYS_TRACE, APP_TRACE, AUDIT };

Detailed Description Paragraph Table (24):
enum DX_INDICATOR { ON, OFF }; class DX_TraceLogObject{ friend class

DX_SysConfigObject; public: //cannot delete the pointer returned //call DeleteInstance( ) static DX_TraceLogObject* Instance (char *componentName, DX_INDICATOR *initTrcLogCatInd, char* initTrcLogDir, long initTrcLogSize); //cannot delete the pointer returned //call DeleteInstance( ) static DX_TraceLogObject* GetInstance( ); static void DeleteInstance( ); //used to write a trace/log message specified by msg //to the //.trc/.log file void Write(char *filename, int lineno, char* context, char* threadId, DX_TL_CATEGORY ctg, char* msg, char* arg1=0, char* arg2=0); //used to write a log message specified by errNo //(access Message Catalog //to get the message) to the.log file void Write(char *filename, int lineno, char* context, char* threadId, DX_TL_CATEGORY ctg, EerrorNumber errNo, char* arg1=0, char* arg2=0); private: static DX_TraceLogObject* instance; //default constructor - does nothing DX_TraceLogObject( ); //constructor used to initialize the Tracing/logging object DX_TraceLogObject(char *componentName, DX_INDICATOR *initTrcLogCatInd, char* initTrcLogDir, long initTrcLogSize); //destructor .about.DX_TraceLogObject( ); //called by sysConfigObject when a deconfigset //command modifies any parameters void ReconfigParameters(char* componentName, DX_INDICATOR *newTrcLogCatInd, char* newTrcLogDir, long newTrcLogSize); //to retrieve the message from the catalog char* GetMessageFromCatalog(int errNum, char* arg1, char* arg2); //to store the settings of the trace/log levels DX_INDICATOR trcLogCategoryInd [DX_NUM_CATEGORIES]; DX_Boolean CheckLogFileSize(ofstream &logStream, char *fileName, long size); void CloseLogFile(ofstream &logStream); DX_Boolean OpenLogFile(ofstream &logStream, char *fileName); //the out streams for the trace and log files ofstream trcStream; ofstream logStream; int logStreamIsOpen; int trcStreamIsOpen; long logSize; char traceFileName[MAX_FILE_NAME]; char logFileName[MAX_FILE_NAME]; DX_Mutex* trcLock; DX_Mutex* logLock; DX_Mutex* trclogParamsLock; }; class DX_TraceLogFormatControl { public: static void WriteFormattedTraceLog (ofstream& outStream, char *category, char *time, char *filename, int lineno, char* threadId, char *message); };

Detailed Description Paragraph Table (25):
Func1( ) { INIT_CONTEXT; ... CONTEXT_BEGIN(Econtext0); // all the trace/log statements in this region will // have the context information // specified by Econtext0. CONTEXT_END; CONTEXT_BEGIN(Econtext1); // in Econtext1 CONTEXT_END; }

CLAIMS:

7. The method of claim 1, further comprising tracking each of the data streams during converting, identifying, or transforming operations.

17. The method of claim 10, further comprising tracking the data during converting, identifying, or transforming operations.

25. The method of claim 19, further comprising: tracking processing of the information having the generic format; and logging errors occurring during the processing of the information having the generic format.

49. The medium of claim 44, further comprising tracking the data during converting, identifying, or transforming operations.

56. The system of claim 51, further comprising means for tracking the data during converting, identifying, or transforming operations.

First Hit Fwd Refs
End of Result Set

Generate Collection Print

L3: Entry 1 of 1 File: USPT Sep 17, 2002

DOCUMENT-IDENTIFIER: US 6453356 B1
TITLE: Data exchange system and method

Detailed Description Text (128):
StringToLog is specified by the developer as a trace message and is written into the <ComponentName><pid>.trc file (type char *). For Example, an AUDIT message could include the details of a Common Object typically stored as a formatted string. ErrorNumber is specified by the developer for a log message (e.g., when the category is CRITICAL, MAJOR or WARNING), and is used to index into a Message Catalog to retrieve the error message to be logged. The error message numbers are defined as an enumerated type as shown below:

Detailed Description Text (216):
Each queue is stored into two categories of files. Each Common Object will be stored as a single file, named by its OID. These files will be evenly distributed, on the modula of 256, into different sub-directories for purposes of even file distribution and performance. These files are generated when the object is serialized. The index of each Common Object is stored in a series of indexed files, indexed from 0 to 9999, which is the physical and persistent storage for the logic queue. Each file may contain up to 100 index records. The order of the index record is defined by the offset of that record to the beginning of the file.

Detailed Description Text (218):
The enqueue operation appends a record at the end of the newest file. The dequeue operation attempts to find a valid record from the internal memory buffer. If one is not found, the dequeue operation will then read one index object from file into the memory buffer. A status field is used to determine the validity of the record. An object can be marked as follows: NEW_OBJECT (object is enqueued but not committed yet); ENQUEUED_OBJECT (object is enqueued and committed); NORMAL_OBJECT (valid object in the queue storage or a the object was rolled back in the memory buffer); ACTIVE_OBJECT (object is read from file into memory buffer and being processed); or DELETED_OBJECT (object has been processed after it is dequeued or it was marked as so in the file). The object in the file storage will only be labeled DELETED_OBJECT after the transaction is committed. An implementation example is given as follows:

62:66
to 64-6

Detailed Description Text (225):
The DX_IndexObject class is important for all the queue operations. It is used to uniquely identify the location at which a Common Object is stored, and is further used for internal routing of all queue requests. The DX_IndexObject class is member of the DX_QueueObject class and DX_QueueOperation class. Its members include queue name, queue priority, storage type, file handle, file index, record offset, and record sequence. An illustrative example of a DX_IndexObject implementation is provided as follows:

67:8 16

Detailed Description Text (230):
The Error Queue is needed to provide storage for run-time processing failures. Since the entries are stored in a queue with an index to the object that generated

the error, the processing logic can decide what further operations should be performed. The implementation of Error Queue is as an instance of DX_Queue. Since no priority issue is involved in Error Queue, the priority argument of the queue operation is relegated to a dummy argument, and only one of the internal child queues need be used.

Other Reference Publication (26):
Product Literature, "MQSeries Business Partners Index,"
http://www.software.ibm.com/ts/mqseries/partners/partner.html, 1 pg. (Feb. 1998).

9/630753

L2: Entry 1 of 1 File: USPT Sep 17, 2002

DOCUMENT-IDENTIFIER: US 6453356 B1
TITLE: Data exchange system and method

Abstract Text (1):
A system and method for exchanging data between two or more applications includes a data exchange engine and a number of adapters associated with a corresponding number of applications. Each of the adapters is customized to interface with a corresponding application and transforms data being transferred between the application and the data exchange engine. Data produced by a particular application is converted from a technology dependent form to a technology independent form by the corresponding adapter. In one embodiment, the format associated with a data stream is disassociated from the informational content of the data stream by the adapter. The informational content of the data stream is then transformed by the adapter into a common or generic format. The data exchange engine receives data in a technology independent form from each of its associated adapters and coordinates the routing of informational content to particular adapters associated with applications that have requested specific informational content. The adapters receiving the informational content from the data exchange engine transform the informational content having the common format into a data format compatible with, or specific to, their associated applications. A queuing mechanism is employed to construct a reliable asynchronous or pseudo-synchronous interface between disparate applications and systems. The data exchange engine may apply business rules or logic when processing a request for particular informational content. User-specified routing logic may be applied by the data exchange engine to dispatch selected informational content to one or more destination applications.

Brief Summary Text (8):
Fourth and fifth generation languages, termed 4GL and 5GL, would appear to offer a partial solution to the data interchange problem. These and other similar languages, such as those used to construct "business objects," are optimized around the construction of applications and user interfaces for the primary purpose of providing powerful querying and reporting tools. The objects created by such languages typically define access paths to data residing in databases. The objects can be combined in various ways to create complex queries and for building powerful report generators. Fourth and fifth generation languages are not well suited for transporting vast amounts of data from one application to one or more other applications in a reliable and efficient manner. Although business objects constructed using 4GL and 5GL techniques do carry with them a certain amount of data, this data is typically data resulting from a query that is transported with an object definition for the purpose of performing additional analysis on the data.

Brief Summary Text (9):
There exists a keenly felt need for a data exchange system and methodology that is capable of exchanging data of varying size, content, and format between dissimilar systems and applications. There exists a further need for such a system and methodology that is independent of any current or future technology. The present invention fulfills these and other needs.

Brief Summary Text (11):
The present invention is directed to a system and method for exchanging data between two or more applications. The data exchange system includes a data exchange engine and a number of adapters associated with a corresponding number of applications. Each of the adapters is customized to interface with a corresponding application and transforms the data being transferred between the application and the data exchange engine. Data produced by a particular application is converted from a technology dependent form to a technology independent form by the corresponding adapter.

Brief Summary Text (12):
In one embodiment, the format associated with a data stream is disassociated from the informational content of the data stream by the adapter. The informational content of the data stream is then transformed by the adapter into a common or generic format. The data exchange engine receives data in a technology independent form from each of its associated adapters and coordinates the routing of informational content to particular adapters associated with applications that have requested specific informational content. The adapters receiving the informational content from the data exchange engine transform the informational content having the common format into a data format compatible with, or specific to, their associated applications. In one embodiment, a queuing mechanism is employed to construct a reliable asynchronous or pseudo-synchronous interface between disparate applications and systems.

Drawing Description Text (15):
FIG. 18 is a class structure diagram showing public and non-public interfaces associated with various file based and database based queuing processes;

Detailed Description Text (119):
A component that intends to record a log/trace message indicates the category to which the message belongs. The log and trace messages are recorded in two different files whose names are derived from the name of the application, as indicated in the following example: (<componentName><pid>.log and <componentName><pid>.trc, respectively)

Detailed Description Text (120):
The possible severity levels for logging various diagnostic messages are as follows. The CRITICAL severity level indicates that the component is in a critical state and cannot proceed properly until the problem is attended to. The MAJOR severity level indicates that a particular activity/operation of the component has failed. However, this may not effect other activities that may continue to run. The WARNING severity level indicates that the component acted in an unexpected manner. This may be something minor, such as a component receiving an invalid message.

Detailed Description Text (128):
StringToLog is specified by the developer as a trace message and is written into the <ComponentName><pid>.trc file (type char *). For Example, an AUDIT message could include the details of a Common Object typically stored as a formatted string. ErrorNumber is specified by the developer for a log message (e.g., when the category is CRITICAL, MAJOR or WARNING), and is used to index into a Message Catalog to retrieve the error message to be logged. The error message numbers are defined as an enumerated type as shown below:

Detailed Description Text (134):
An error/event may occur at a very low level in the code (e.g., database space exhausted). It is important to report this low level event, but it is also important to report the context of what was trying to be achieved within the application when this low level error occurred. The application developer is provided with macros to define a context within the developer's code. The set of

macros provided for this purpose include: INIT_CONTEXT; CONTEXT_BEGIN; and CONTEXT_END. In general, every function using the context macros should first use the macro INIT_CONTEXT. It is noted that, if INIT_CONTEXT is not called before defining CONTEXT_BEGIN, the code may not compile.

Detailed Description Text (135):
The beginning of a context may be defined using the macro CONTEXT_BEGIN, and the end of a context can be defined using the macro CONTEXT_END, as is indicated in the following example. The CONTEXT_BEGIN macro takes the argument Context Number. This context number is used to access the Context Catalog of an application and to retrieve the context string. It is noted that nested contexts are generally not allowed. If a CONTEXT_BEGIN is called before the previous context is ended, an implicit CONTEXT_END for the previous context is assumed. The following example is provided:

Detailed Description Text (147):
The following statistics are typically recorded when monitoring is performed on the queue: (1) number of the messages processed in the system input queue; (2) the average message cache time in the system input queue, by priority; and (3) the average message processing time from the system input queue, by priority. The following statistics are typically recorded when monitoring is performed on the disk space and database table space usage: (1) the percentage of the disk space usage if the queue storage type is FLATFILE; and (2) the percentage of the table space usage if the queue storage type is DATABASE.

Detailed Description Text (151):
The monitor writes the log file in the same format irrespective of whether the queue is file based or database based. The format of the report file is provide in Table 1 below:

Detailed Description Text (153):
When disk space and/or database table space usage is being monitored (i.e., SYSTEM_INFO_MONITOR=ON), the monitor writes the usage of the disk where the queue files located into an ASCII file on a regular time interval if the queue is file based. The name of the file follows the system file name schema (e.g., <AppName>_SysInfo.Mon_Log). The maximum file size is defined in the system configuration file. After the file reaches its maximum size, the monitor moves it to a backup copy named <AppName>_SysInfo.Mon_Log.bk and creates and writes the performance into the new <AppName>_SysInfo.Mon_Log file. The system retains only one backup copy of the monitor log files. The format of the report file is give below in Table 2:

Detailed Description Text (154):
where, Time represents the time stamp of the record in the log file; and DiskUsage represents the percentage of the file disk usage if STORAGE_TYPE=FLATFILE. Again, the data fields are delimited by a comma. It is noted that the database table space usage is generally not available to the application user, such that only the system administrator has the privilege to view it. As such, the monitor does not perform monitoring for the database table space usage.

Detailed Description Text (157):
The Monitor Object is instantiated by the DX_SysConfigObject instance or by calling the static method DX_Monitor::Instance( ) directly. There is only one DX_Monitor thread running per executable component. The monitor thread is spawned whenever the MONITOR in the system configuration file is triggered to ON. The monitor thread exists until the MONITOR is triggered to OFF. The implementation of the monitor impacts three areas. The Queue Manager provides the queue performance data. The DX_SysConfigObject provides the configuration change handling and the monitor object instantiation. The DX_Monitor Object spawns or kills the monitor and generates the monitor log files or log table in the database. The methods added in

the DX_Queue classes are listed below:

Detailed Description Text (168):
A shell/bat script DX_Shutdown is provided to shutdown individual components identified by the component name. DX_Shutdown needs to halt all the threads of execution gracefully before shutting down the component. The syntax of the DX_Shutdown script is DX_Shutdown <ComponentName><Pid>. An implementation example is provided as follows. DX_Shutdown script uses DX_ConfigSet to communicate with the component being shutdown. DX_ConfigSet can be used to raise a signal or event after adding a parameter to the config file to indicate that an instance of the particular component is to shut down gracefully. The configuration parameter used is DX_SHUTDOWN and its value is set to the PID of the instance to be shutdown.

Detailed Description Text (177):
Concerning data exchange system security, the basic security control is focused on the queue files access or the database tables access. The file access control requires the application user to be in a specific user group. The user group should be set before the application runs. The database table access control requires the application users to have the correct user name and user password. The user name and user password may be set in the environment variables or be hard coded in the application programs. In one embodiment, all applications share one database user account. This user account has privileges to create/update/delete tables in the view.

Detailed Description Text (183):
A DX_Utils library provides the DX_Mutex class for platform independent mutex protection, an example of which is provided below. The DX_Mutex class does not require use of the DX_ThreadController class.

Detailed Description Text (187):
A skeleton main( ) function is provided to illustrate the system initialization and startup procedures required for each component of the data exchange system application. This includes operations such as database connection, system resource configuration, thread control, etc. In addition, a System Health Monitor thread is provided which, on a timed interval, polls all system resources to ensure that system operation can continue. This thread invokes the system checking operations System Configuration Object. The sample code provided below illustrates the ease of initializing system components and application operations. This sample is part of the DX_Engine executable, which serves as the core of the data exchange system.

Detailed Description Text (196):
A further description of a data exchange system queuing methodology in accordance with one embodiment of the present invention will now be described. In order to provide a clean "buffered API," a queued request approach is used. Use of interface queues allows the caller of the API to send its request irrespective of whether the engine core and another outgoing adapter are running. The queue interface approach of the instant embodiment also provides a mechanism for buffering the load that may be placed upon a server from multiple clients. It also provides the ability to scale the number of database servers that can process any given queue in parallel.

Detailed Description Text (197):
As was discussed previously, two types of priority based queues are used, namely, the incoming Receive Queues and the outgoing Send Queues. Each outgoing adapter will have its own outgoing queue so that any interface specific translation or routing may be performed outside the engine core. Each instance of the DX_Engine executable has one or more input queues, although only one is allowed for file-based queues, and one or more output queues. An instance of the DX_QueueManager class is used as a central proxy to all queue access, and will be mutex protected and record-lock protected, for file-based implementation, or row lock protected, for database implementations, to prevent data contention.

Detailed Description Text (198):
Two types of queues are provided, file and database queues, both of which are fairly simple implementations that allow for a clean breakup to the API. Priority based queuing is provided so that requests of high importance can be serviced quickly. Multiple queues are used to provide this level of functionality, but the implementation is logically transparent to users. A user perceives that there is only one logic queue with objects of different priority in it.

Detailed Description Text (199):
The file storage or database tables for the queue are created at running time and deleted by queue administration process. There are four types of pre-defined priority: NONURGENT; NORMAL; URGENT; and INTERACTIVE in order of increasing priority. INTERACTIVE is the highest priority, which can block any request having other priorities. The priority algorithm ensures that the Dequeue operation always returns successfully when the queue is not empty, prevents starvation of lower priority entries, and ensures more frequent visits on higher priority queues. The priority algorithm is implemented on a weighted basis of each priority.

Detailed Description Text (200):
Support for parallel gateways is only available to a database queue implementation. File-based queue implementations will not guarantee single delivery, i.e., one object might be dequeued by multiple process at the same time. All parallel access should be completely transparent to any participating gateway. The only areas of common resources between any parallel gateways are the Enqueue and Dequeue operations. The design of the Enqueue/Dequeue module ensures that parallel access is made possible without any deadlocks or duplicated queue entries by using the database supplied row-level locking.

Detailed Description Text (201):
Since the external API is limited to the Enqueue/Dequeue API, the only limit to multiple access is the row-level table locking that the database supports. The file based queue mechanism uses simple file record-lock to protect from multiple updates to the file from multiple threads. The queue access operations for file-based implementation are thread-safe, such that all the operations are mutex protected.

Detailed Description Text (202):
The Queue Manager public interface makes use of the DX_QueueTransaction object for transaction control. The Enqueue( ), Dequeue( ), Commit( ), and Rollback( ) methods take pass-in argument of an instance of the DX_QueueTransaction class which belongs to a running thread. The transaction object contains an ordered list of operations performed in this transaction. For file-based implementations, all operations are maintained in buffered memory and are not written into file storage until commit time. For database implementations, the database provided rollback mechanism is employed, with each transaction using its own unique run-time database context.

56:3-8

Detailed Description Text (212):
DX_FileQueue class contains four child queues for each priority. Besides the queue operation interface and transaction interface, the algorithm of priority handling is also implemented in this class. The priority algorithm is implemented inside the Dequeue method. The DX_IndexObject argument of the Dequeue methods is used for transaction control. At running time, Dequeue operations fill in corresponding fields in DX_IndexObject, which is a component of DX_QueueOperation object. An implementation example is given as follows:

Detailed Description Text (216):
Each queue is stored into two categories of files. Each Common Object will be stored as a single file, named by its OID. These files will be evenly distributed, on the modula of 256, into different sub-directories for purposes of even file distribution and performance. These files are generated when the object is

serialized. The index of each Common Object is stored in a series of indexed files, indexed from 0 to 9999, which is the physical and persistent storage for the logic queue. Each file may contain up to 100 index records. The order of the index record is defined by the offset of that record to the beginning of the file.

Detailed Description Text (218):
The enqueue operation appends a record at the end of the newest file. The dequeue operation attempts to find a valid record from the internal memory buffer. If one is not found, the dequeue operation will then read one index object from file into the memory buffer. A status field is used to determine the validity of the record. An object can be marked as follows: NEW_OBJECT (object is enqueued but not committed yet); ENQUEUED_OBJECT (object is enqueued and committed); NORMAL_OBJECT (valid object in the queue storage or a the object was rolled back in the memory buffer); ACTIVE_OBJECT (object is read from file into memory buffer and being processed); or DELETED_OBJECT (object has been processed after it is dequeued or it was marked as so in the file). The object in the file storage will only be labeled DELETED_OBJECT after the transaction is committed. An implementation example is given as follows:

Detailed Description Text (223):
DX_DBQueue is the database-based implementation of the DX_Queue interface. Its instance is mapped to a single table per queue at run-time. The order of the records is determined by the enqueue time stamp. All dequeue operations are sorted by priority and enqueue time stamp. An illustrative example of a DX_DBQueue implementation is provided as follows:

Detailed Description Text (225):
The DX_IndexObject class is important for all the queue operations. It is used to uniquely identify the location at which a Common Object is stored, and is further used for internal routing of all queue requests. The DX_IndexObject class is member of the DX_QueueObject class and DX_QueueOperation class. Its members include queue name, queue priority, storage type, file handle, file index, record offset, and record sequence. An illustrative example of a DX_IndexObject implementation is provided as follows:

Detailed Description Text (227):
The invocation of the Enqueue( ) and Dequeue( ) API is effected by sending a request to the Queue Manager Object. In response to a request, the Queue Manager Object locates the correct queue and populates the operations to that queue object. When Enqueue( ) is invoked, the object is placed into a buffer list and will not be collapsed into a data stream until Commit( ) is invoked. Until the commit time, the DX_IndexObject attribute of the DX_QueueOperation object retains the real meaning, which may be used in connection with a rollback operation if the commit operation is not successfully executed. For purposes of serialization, if the queue is a database, the row-level locking provided by the database is used. If the queue is a file, file access control is used. When a Common Object is serialized at commit time, a priority tag is appended to its private attribute list, such that when the Common Object is dequeued, its priority can be easily determined.

Detailed Description Text (228):
When Dequeue( ) is invoked, the oldest entry with the proper priority is retrieved with the marshalled object then being instantiated as a Common Object using the Demarshal( ) method invocation. This mechanism provides database row-locking on read events to prevent parallel gateways reading the same queued requests. If the queue is file based, record lock is used. The logic to determine which priority queue should be invoked on implemented inside the DX_Queue object.

Detailed Description Text (230):
The Error Queue is needed to provide storage for run-time processing failures. Since the entries are stored in a queue with an index to the object that generated

the error, the processing logic can decide what further operations should be performed. The implementation of Error Queue is as an instance of DX_Queue. Since no priority issue is involved in Error Queue, the priority argument of the queue operation is relegated to a dummy argument, and only one of the internal child queues need be used.

Detailed Description Text (232):
Objects are stored and retrieved from a persistent store in an efficient manner. Two types of object storage, termed relational database storage and file storage, are implemented for this purpose. The object persistency is implemented using Marshal( ), Demarshal( ), and DeleteStorage( ) methods of the DX_CommonObject class, where a parameter is passed to indicate storage type. A policy for object caching and sharing may also be employed.

Detailed Description Text (233):
Since database implementation may vary among venders and file based persistency is needed, the object persistency model has been developed to be independent from any database storage technology. This approach involves collapsing the object into a raw binary data stream, and uses various headers to denote any marker and priority related information. The headers of a Common Object typically include OID and priority tag information. Table 4 provide below illustrates a how a Common Object may be stored in a typical database.

Detailed Description Text (234):
As stated previously, all database access is hidden inside the Enqueue( ) and Dequeue( ) methods of DX_QueueManager and Marshal( ), Demarshal( ), and DeleteStorage( ) methods of DX_CommonObject. In one embodiment, all vendor specific access mechanisms may be delivered in a separate set of database enabled libraries.

Detailed Description Text (238):
File-based database access is invoked using the Marshal( ), Demarshal( ), and DeleteStorage( ) methods from the Common Object with the output argument set to a file. Each object may be stored to a separate file where the filename incorporates the object ID. By using the object counter mechanism as a directory name, files can be spread evenly across a file system for better access time.

Detailed Description Text (240):
A transaction object should be created and destroyed as a local object of a thread. If neither Commit( ) nor Rollback( ) was called when a thread exits, all operations executed by this thread are rolled back. For a database implementation, the native transaction control mechanism of the database is used. For a file implementation, a transaction object contains a list recording of all operations in the current transaction. This memory buffer is used to buffer all queue operations and will not be written into file storage until commit time. Rollback( ) removes the operations from that memory buffer for operations that have not been committed. Since Commit ( ) might fail, Rollback( ) also cleans the entries in the queue files and Common Object file for those operations that failed at commit time.

Detailed Description Paragraph Table (2):
enum EclassTypes { DX_OBJECT = 0, DX_ATTRIBUTE, DX_LIST, DX_MULTIVALUE, DX_INTEGER, DX_STRING, DX_STRINGVAL, DX_DATE, DX_TIME, DX_REAL, DX_BLOB, DX_BLOBVALUE, DX_VERSION, DX_OID_CLASS }; enum EreturnCodes { NOT_FOUND = 0, SUCCESS, FAILED, TIME_OUT, ACCESS_DENIED, DUPLICATE_OBJECT, NO_OBJECT, NO_ATTRIBUTE, INVALID_ATTRVAL, INVALID_ARGS, INVALID_OPERATION, SYSTEM_ERROR }; enum EstorageTypes { FLATFILE = 0, DATABASE }; enum ElistType { PUBLIC = 0, PRIVATE }; static const int MAX_NAME_LEN=255; static const int MAX_DOTTED_NAMELEN=8096; static const int MAX_MULTI_VAL=255; static const int MAX_BLOB_SIZE=2048; static const int OID_LEN=128; static const int MAX_FILE_NAME=256; static const int MAX_PID_LEN=12; static const int MAX_LINE_SIZE=100;

Detailed Description Paragraph Table (3):

```
class: DX_CommonObject: public DX_CommonBase { RWDECLARE_COLLECTABLE
(DX_CommonObject); friend class DX_ListObject; friend class DX_FileSubQueue;
public: virtual.about.DX_CommonObject(); //
Constructors /*****************************************************************/ //I
name==NULL, the name is set to "NOT_SET". //If name is ""or contains a "." it will
be set to
"INVALID_NAME" /*****************************************************************/
DX_CommonObject(const char*
name=0); /*****************************************************************/ //Creat
a copy of an entire DX_CommonObject, //but with it's own unique OID
value /*****************************************************************/
DX_CommonObject* Clone
(); /*****************************************************************/ //All
AddAttribute and AddPvtAttribute methods take ownership of //the pointers passed in
to them. Do NOT delete the pointers after //a call to AddAttribute and
AddPvtAttribute. The pointers will be //deleted when the container DX_CommonObject
or DX_ListObject is deleted. // //NOTE: When using the following two methods for
creating a DX_STRING attribute // // AddAttribute(const char* name, const int type,
const void* value) // and AddPvtAttribute(const char* name, const int type, const
void* value) // // the value is defaulted to be of type const char* and not
UNICHAR* // Misuse will lead to unexpected
behavior. /*****************************************************************/
EreturnCodes AddAttribute(DX_CommonAttribute* newAttr); EreturnCodes AddAttribute
(DX_ListObject* newAttr); EreturnCodes AddAttribute(DX_CommonObject* newAttr);
EreturnCodes AddAttribute(const char* name, const int type, const void* value);
EreturnCodes AddPvtAttribute(DX_CommonAttribute* newAttr); EreturnCodes
AddPvtAttribute(DX_ListObject* newAttr); EreturnCodes AddPvtAttribute
(DX_CommonObject* newAttr); EreturnCodes AddPvtAttribute(const char* name, const
int type, const void*
value); /*****************************************************************/ //Delete
and DeletePvtAttribute will remove the named attribute //from the container
DX_CommonObject or DX_ListObject and delete the named //attribute's
pointer /*****************************************************************/
EreturnCodes DeleteAttribute(const char* name); EreturnCodes DeletePvtAttribute
(const char*
name); /*****************************************************************/ //RemoveA
will remove the named attribute from the container //DX_CommonObject or
DX_ListObject but will not delete the named //attribute's
pointer /*****************************************************************/
EreturnCodes RemoveAttribute(const char* name); EreturnCodes RemovePvtAttribute
(const char*
name); /*****************************************************************/ //Do NOT
delete the pointer that is returned to you, it still is owned by //the container
DX_CommonObject or DX_ListObject. You CAN use the any of //the attribute class
methods for the
pointer. /*****************************************************************/ void*
GetAttribute(const char* name); void* GetPvtAttribute(const char* name); void
PrintContents(); //The caller of Demarshal is responsible for object's memory
allocation static DX_CommonObject* Demarshal(char* ObjOid, int type, int
ContextIndex); EreturnCodes Marshal(int type, int ContextIndex); static
EreturnCodes DeleteStorage(const char* oidVal, int type, int ContextIndex); //The
caller of GetOID is responsible for deleting the memory of the returned char* char*
GetOID(); EPriorityCode GetPriority(); protected: void* Find(const char* name);
static EreturnCodes RestorePersistentObject(const char* oidVal); static
EreturnCodes DeletePersistentObject(const char* oidVal, int type); private: //Data
Members DX_ListObject* PrivateList; DX_ListObject* PublicList; static void Delete
(RWDlistCollectables* list); static int Copy(RWDlistCollectables* dest, const
RWDlistCollectables *src); static EreturnCodes CopyPersistentObject(char*
```

filename); static EreturnCodes CopyFile(char* filename,char* copyfilename); void saveGuts(RWFile& file) const; void saveGuts(RWvostream& stream) const; void restoreGuts(RWFile& file); void restoreGuts(RWvistream& stream); const int check() const; DX_Boolean updateListOids(DX_ListObject* Plist, DX_OID *Poid); DX_Boolean updateObjectOids(DX_CommonObject* Pobj, DX_OID *Poid); DX_Boolean ValidateNaming (const char* name); #ifdefDX_DATABASE EreturnCodes InsertIntoTable(char* filename, int ContextIndex); static EreturnCodes RetrieveFromTable(char* filename, char* oid, int ContextIndex); EreturnCodes UpdateTable(char* filename, int ContextIndex); #endif //use to cal. the number of bytes needed to store an object using RWFile RWspace binaryStoreSize() const;}; };

Detailed Description Paragraph Table (34):
class: DX_QueueLogMgr { public: // create an instance of the object by calling the constructor // delete using DeleteInstance( ) static DX_QueueLogMgr* Instance(char* ComponentName=0); // delete the object static void DeleteInstance( ); // Checks if the QueueStatusList has an entry // for the particular queue and if so checks // if the entry indicates whether the // queue monitoring is on or off and logs the // QueueObject accordingly static void DumpQLog(EQueueOperation op, char* queueName, DX_QueueObject* qo); // if you want to monitor a queue, an entry in // the queue status list should be created first // by giving the queue name and the pid. The function // reads the .reg file and initializes the entry // accordingly. EreturnCodes InsertQueueStatusList(char* QName, int pid); // if queue monitor changed the queue registration file, reset the // object, read the reg file and update the queueStatusList; EreturnCodes ReconfigQueueStatusList( ); private: DX_Mutex* listLock; static DX_QueueLogMgr* instance; char ComponentName [MAX_NAME_LEN]; char QlogDir[MAX_FILE_NAME]; DX_QueueLogMgr(char* inComponentName=0,char* qlogdir=0); virtual.about.DX_QueueLogMgr( ); EreturnCodes DeleteFromRegFiles(char* ComponentName,int pid); DX_INDICATOR GetStatusFromRegFile (char* regFileName, int& regFileExists); char* GetQRegFileName(char* QName); EreturnCodes FindQueueLogStatusInfo(char* inQName, DX_QueueLogStatusInfo** copy); RWDlistCollectables *queueStatusList; }; class DX_QueueLogStatusInfo: public RWCollectable { friend class DX_QueueLogMgr; private: char qName[MAX_NAME_LEN]; DX_INDICATOR qLogStatus; char qLogFileName[MAX_FILE_NAME]; RWDECLARE_COLLECTABLE (DX_QueueLogStatusInfo); DX_QueueLogStatusInfo( ){ }; // During construction if inQLogStatus is ON // open the logStream for QName.qlog in directory DX_HOME/DX_QLOG DX_QueueLogStatusInfo(char* inQName, DX_INDICATOR inQLogStatus, char* qlogdir); // close all the streams which are open .about.DX_QueueLogStatusInfo( ); char* GetQName( ); DX_INDICATOR GetQLogStatus ( ); char* GetQLogFileName( ); // if set to on, open stream, if set to off close the stream EreturnCodes SetQLogStatus(DX_INDICATOR inStat); }; #define DX_QLOG (QueueOp,QueueName,qo) DX_QueueLogMgr::.backslash. DumpQLog(QueueOp,QueueName,qo);

Detailed Description Paragraph Table (42):
enum EstorageTypes { FLATFILE = 0, DATABASE }; enum EQueueOperation { ENQUEUE = 0, DEQUEUE }; static const int NUM_PRIORITIES = 4; enum EPriorityCode { NONURGENT = 0, NORMAL, URGENT, INTERACTIVE };

Detailed Description Paragraph Table (43):
class: DX_QueueManager { friend class DX_QueueTransaction; friend class DX_DBQueue; friend class DX_Monitor; public: static DX_QueueManager* Instance(const char* ProcessName, EstorageTypes type); static void DeleteInstance( ); static DX_QueueManager* GetInstance
( ); /*****/ //Queue operation
interface /*****/ /******
label and comment arguments will be used for Queue Administration Purpose. //So queue administration GUI will also see the name and comment of each CO in the //queue. They can be type of UTF-8 encoded string, 7-bit ASCII string or wide string. //User should not delete pointer to DX_CommonObject after call Enqueue /*****/

```
EreturnCodes Enqueue(const char* qName, DX_CommonObject &co, DX_QueueTransaction
&transaction, const char* oLabel, const char* comment, EPriorityCode priority =
NORMAL); EreturnCodes Enqueue(const char* qName, DX_CommonObject &co,
DX_QueueTransaction &transaction, const UNICHAR* oLabel, const UNICHAR* comment,
EPriorityCode priority =
NORMAL); /*****************************************************************/ //Caller
should free non-NULL pointer to DX_CommonObject //Return SUCCESS if Dequeue
returned a common object //Return FAILED if Dequeue returned a NULL common object
pointer /*****************************************************************/
EreturnCodes Dequeue(const char* qName, DX_CommonObject* &CO DX_QueueTransaction
&transaction); EreturnCodes Dequeue(const char* qName, const char* objID, const
char* objLabel, DX_CommonObject* &co, DX_QueueTransaction
&transaction); /*****************************************************************/ //
caller of GetCursor is responsible for deleting //returned DX_IndexObject*
pointer. /*****************************************************************/
DX_IndexObject* GetCursor(const char* qName, EPriorityCode priority =
INTERACTIVE); /*****************************************************************/ //W
set the cursor to the EPriorityCode passed
in /*****************************************************************/ EreturnCodes
ResetCursor(DX_IndexObject &cursor, EPriorityCode priority =
INTERACTIVE); /*****************************************************************/ //T
always is a non-null DX_QueueList returned //Caller should delete
DX_QueueObjectList returned // //NOTE: DX_QueueList may be empty if no entries were
found // //USAGE: //GetQueueView(DX_QueueObjectList* &list DX_IndexObject
&QViewCursor, // int size = 0) //will return #entries =< size for EPriorityCode of
QViewCursor and ALL lower priorities // //GetQueueView(DX_QueueObjectList* &list,
EPriorityCode priority, // DX_IndexObject &QViewCursor, int size = 0) //will return
#entries =< size for EPriorityCode of priority, QViewCursor is updated to //reflect
position of last retrieved
entry. /*****************************************************************/
EreturnCodes GetQueueView(DX_QueueObjectList* &list, DX_IndexObject &QViewCursor,
int size = 0; EreturnCodes GetQueueView(DX_QueueObjectList* &list, EPriorityCode
priority, DX_IndexObject &QViewCursor, int size =
0); /*****************************************************************/ //Caller
should free char** returned
twice /*****************************************************************/ char**
GetManagedQueueNames(int& number); char** GetAllQueueNames(int& number); private:
static DX_QueueManager* instance; char* owner; EstorageTypes Implementation; char*
FileDBDirectory; RWGDlist(DX_Queue) QueueList; DX_Mutex* mutex; DX_QueueManager
(const char* processID, EstorageTypes type, const char* FileDBDir);
.about.DX_QueueManager( ); //Extended transaction support interface EreturnCodes
Commit(DX_QueueTransaction &transaction); EreturnCodes Rollback(DX_QueueTransaction
&transaction); //Queue administration interface EreturnCodes DeleteQueue(const
char* qName); EreturnCodes FlushQueue(const char* qName); EreturnCodes FlushQueue
(const char* qName, EPriorityCode priority); EreturnCodes RemoveFromQueue
(DX_QueueObject *object); //Performance monitor interface //the memory of
pNumOfMsgProcessed, pAvgMsgCacheTime, pAvgMsgProcessTime //should be allocated
before invoking this method. EreturnCodes GetQueuePerformance(char * inputQName,
long *pNumOfMsgProcessed, double *pAvgMsgCacheTime, double *pAvgMsgProcessTime,
DX_Boolean resetFlag = TRUE); void ResetAll(const char *qNameList); EreturnCodes
GetDBTableSpaceUsage(float &usage); //Internal use DX_Queue* FindQueue(const char*
qName); DX_Queue* CreateQueue(const char* qName); };
```

Detailed Description Paragraph Table (44):

```
class: DX_Queue { friend class DX_QueueManager; friend DX_Boolean IsQueueEqual
(const DX_Queue* queue, const void* value); protected: virtual .about. DX_Queue( );
virtual EreturnCodes Enqueue(DX_CommonObject& co, const char* ProcessId, const
char* label, const char* comment, DX_QueueTransaction& transact, EPriorityCode
pCode) = 0; virtual DX_CommonObject* Dequeue(DX_QueueTransaction& transact) = 0;
virtual DX_CommonObject* Dequeue(DX_QueueTransaction& transact, const char*
```

ObjectID, const char* ObjectLabel) = 0; //Only DX_FileQueue need to implemented the following two methods virtual EreturnCodes Commit(DX_QueueOperation& oper) {return FAILED;} virtual EreturnCodes CompleteCommit(DX_QueueOperation& oper) {return FAILED;} virtual EreturnCodes Rollback(DX_QueueOperation& oper) {return FAILED;} const char* GetQueueName( ) const; virtual EreturnCodes DestroyStorage( ) = 0; virtual EreturnCodes Flush(EPriorityCode priority) = 0; virtual EreturnCodes RemoveObject(DX_QueueObject* qObj) = 0; virtual EreturnCodes GotoBeginning (EPriorityCode priority, DX_IndexObject &cursor) = 0; virtual EreturnCodes GetQueueView(EPriorityCode priority, DX_IndexObject &cursor, DX_QueueObjectList* &list, int size) = 0; //performance monitor virtual void GetQueuePerformance(long *pNumObjectProcessed, double *pAvgMsgCacheTime, double *pAvgMsgProcessTime) = 0; virtual void Reset( ) = 0; //We should not have instance of this class DX_Queue (const char* qName); in line void SetWeightRootsAndVisitedFlags( ) { roots [NONURGENT] = 0.0f; roots[NORMAL] = 0.6f; roots[URGENT] = 0.8f; roots[INTERACTIVE] = 1.0f; //will always block other priorities VisitedFlags[NONURGENT] = 0x01; VisitedFlags[NORMAL] = 0x02; VisitedFlags[URGENT] = 0x04; VisitedFlags[INTERACTIVE] = 0x08; } //Because these members should be seen by the derived classes, //we keep them as protected. float roots[NUM_PRIORITIES]; unsigned char VisitedFlags [NUM_PRIORITIES]; float weights[NUM_PRIORITIES]; char *QueueName; };

Detailed Description Paragraph Table (45):
class: DX_FileQueue : public DX_Queue { friend class DX_QueueManager; private: DX_FileQueue(const char* qName, const char* FileDBDir); .about.DX_FileQueue( ); EreturnCodes Enqueue(DX_CommonObject& co, const char* ProcessId, const char* label, const char* comment, DX_QueueTransaction& transact, EPriorityCode pCode); DX_CommonObject* Dequeue(DX_QueueTransaction& transact); DX_CommonObject* Dequeue (DX_QueueTransaction& transact, const char* ObjectID, const char* ObjectLabel); EreturnCodes Commit(DX_QueueOperation& oper); EreturnCodes CompleteCommit (DX_QueueOperation& oper); EreturnCodes Rollback(DX_QueueOperation& oper); EreturnCodes DestroyStorage( ); EreturnCodes Flush(EPriorityCode priority); EreturnCodes RemoveObject(DX_QueueObject* qObj); EreturnCodes GotoBeginning (EPriorityCode priority, DX_IndexObject &cursor); EreturnCodes GetQueueView (EPriorityCode priority, DX_IndexObject &cursor, DX_QueueObjectList* &list, int size); //used for the performance monitor void GetQueuePerformance(long *numOfMsgProcessed, double *avgMsgCacheTime, double *avgMsgProcessTime); void Reset ( ); private: DX_FileSubQueue* subqueues[NUM_PRIORITIES]; DX_Mutex* WeightMutex; DX_Mutex* SubQueueMutex[NUM_PRIORITIES]; };

Detailed Description Paragraph Table (46):
class: DX_FileSubQueue { friend class DX_FileQueue; friend class DX_QueueManager; private: DX_FileSubQueue(const char* qName, EPriorityCode pCode, const char* FileDBDir); .about.DX_FileSubQueue( ); EreturnCodes Enqueue(DX_CommonObject& co, const char* ProcessId, const char* label, const char* comment); DX_CommonObject* Dequeue(DX_IndexObject& io); DX_CommonObject* Dequeue(DX_IndexObject& io, const char* ObjectID, const char* ObjectLabel); EreturnCodes Commit(DX_QueueOperation& oper); EreturnCodes CompleteCommit(DX_QueueOperation& oper); EreturnCodes Rollback (DX_QueueOperation& oper); EreturnCodes DestroyStorage( ); EreturnCodes Flush( ); EreturnCodes RemoveObject(DX_QueueObject* qObj); EreturnCodes GotoBeginning (DX_IndexObject& cursor); EreturnCodes GetQueueView(DX_IndexObject &cursor, DX_QueueObjectList* &list, int size); //used by the performance monitor long GetNumOfMsgProcessed( ); double GetTotalMsgCacheTime( ); double GetTotalMsgProcessTime( ); void Reset( ); private: char* QueueName; EPriorityCode Priority; char* QueueFileName; char* IndexDirectory; int startFileIndex; int endFileIndex; //These two fields are used for Dequeue operation and always go forward int currentFileIndex; int currentRecordIndex; int lastRecordIndex; DX_QueueObjectList BufferList; ////////////////////////// //Internal use only ////////////////////////// EreturnCodes EnqueueCommit(DX_QueueOperation &oper); EreturnCodes DequeueCommit(const char* oid, const DX_IndexObject* io); EreturnCodes CompleteEnqueueCommit(DX_QueueOperation &oper); EreturnCodes CompleteDequeueCommit(const char* oid, const DX_IndexObject* io); EreturnCodes

EnqueueRollback(const char* oid, const DX_IndexObject* io); EreturnCodes DequeueRollback(const char* oid, const DX_IndexObject* io); EreturnCodes TryRecycleFile(int fIndex); void TryRecycleQueue( ); char* GetFileName(int fIndex); int OpenFile(int whichFile); EreturnCodes CreatEndFile( ); void UpdateIndexFile( ); void UpdateFileIndex( ); EreturnCodes ExpandQueue( ); EreturnCodes MarkObjectInFile (const DX_IndexObject* iObj, EQueueObjectStatus status); //Unix does not have low-level eof IO function available int IsEOF(int fd); //used by performance monitor double totalMsgCacheTime; double totalMsgProcessTime; long numOfMsgCommited; void SetDequeueTime(DX_QueueObject *qo); void SetEnqueueTime(DX_QueueObject *qo); void CompleteCommitCalculation(DX_QueueObject *qo); };

Detailed Description Paragraph Table (47):
class: DX_DBQueue : public DX_Queue { friend class DX_QueueManager; private: DX_DBQueue(const char* qName); .about.DX_DBQueue( ); EreturnCodes Enqueue (DX_CommonObject& co, const char* ProcessId, const char* label, const char* comment, DX_QueueTransaction& transaction, EPriorityCode priority); DX_CommonObject* Dequeue(DX_QueueTransaction &); DX_CommonObject* Dequeue (DX_QueueTransaction& transaction, int priority); DX_CommonObject* Dequeue (DX_QueueTransaction& transaction, const char* ObjectID, const char* ObjectLabel); //Virtual methods inherited from DX_Queue and NOT USED EreturnCodes Commit(DX_QueueOperation& oper) {return FAILED;} EreturnCodes CompleteCommit (DX_QueueOperation& oper) {return FAILED;} EreturnCodes Rollback(DX_QueueOperation& oper) {return FAILED;} static EreturnCodes Commit(DX_QueueTransaction& transaction); static EreturnCodes Rollback(DX_QueueTransaction& transaction); static EreturnCodes CreateTable(int DBcontextId, const char* qName); EreturnCodes DestroyStorage( ); EreturnCodes Flush(EPriorityCode priority); EreturnCodes RemoveObject(DX_QueueObject* qObj); EreturnCodes GotoBeginning(EPriorityCode priority, DX_IndexObject &cursor){return SUCCESS;} EreturnCodes GetQueueView (EPriorityCode priority, DX_IndexObject &cursor, DX_QueueObjectList* &list, int size); //used for the performance monitor void GetQueuePerformance(long *NumOfMsgProcessed, double *avgMsgCacheTime, double *avgMsgProcessTime); void Reset ( ); static EreturnCodes GetTableSpaceUsage(float &usage); static char** GetAllQueueNames(int& number); //Data member DX_Mutex* WeightMutex; //used by performance monitor long *numOfMsgCommited; double *totalMsgCacheTime; double *totalMsgProcessTime; static void CommitCalculations(DX_QueueTransaction &transaction); };

Detailed Description Paragraph Table (48):
class: DX_IndexObject { friend class DX_QueueObject; friend class DX_QueueOperation; friend class DX_FileSubQueue; friend class DX_FileQueue; friend class DX_DBQueue; friend class DX_QueueManager; public: //Because GetQueueView needs an instance of DX_IndexObject as cursor //and user should be able free the instance after use .about.DX_IndexObject( ); private: DX_IndexObject(EstorageTypes type); DX_IndexObject(EstorageTypes type, const char *qName, EPriorityCode priority); DX_IndexObject(const DX_IndexObject& inputIO); DX_IndexObject& operator= (const DX_IndexObject& inputIO); in line EstorageTypes GetStorageType( ) const { return StorageType; } in line const int GetFileHandle( ) const { return FileHandle; } in line const char* GetQueueName( ) const { return QueueName; } in line EPriorityCode GetPriority( ) const { return Priority; } in line int GetFileIndex( ) const { return FileIndex; } in line long GetRecordOffset( ) const { return RecordOffset; } in line long GetTimeStamp( ) const { return TimeStamp; } EreturnCodes SetFileHandle(int fh); EreturnCodes SetQueueName(const char* qName); EreturnCodes SetPriority(EPriorityCode priority); EreturnCodes SetRecordOffset(long rOffset); EreturnCodes SetFileIndex(int fIndex); EreturnCodes SetTimeStamp(long tm); private: char* QueueName; EPriorityCode Priority; EstorageTypes StorageType; int FileHandle; //handle to a file already opened int FileIndex; long RecordOffset; long TimeStamp; //enqueue time };

Other Reference Publication (23):
Product Literature, "SAIC The Vision," http://www.saic.com/telecom/index.html, 4

pgs. (Jan. 1998).

Other Reference Publication (26):
Product Literature, "MQSeries Business Partners Index,"
http://www.software.ibm.com/ts/mqseries/partners/partner.html, 1 pg. (Feb. 1998).

CLAIMS:

1. A method of transporting data, comprising: receiving a data stream from each of a plurality of source applications, each of the data streams comprising informational content and having a technology dependent form associated with a source protocol; converting the data streams from the technology dependent forms to technology independent forms not associated with the respective source protocols and not associated with respective destination protocols of one or more destination applications; identifying the one or more destination applications; transporting the data streams having the technology independent forms; transforming the data streams from the technology independent forms to technology dependent forms associated with the respective destination protocols of the one or more of the destination applications; and transmitting all or a portion of the data streams having the technology dependent forms to the one or more of the destination applications.

10. A method of transporting data, comprising: receiving, from a source application, data comprising informational content in a technology dependent form associated with a source protocol; converting the data from the technology dependent form associated with the source application to a technology independent form not associated with the source protocol and not associated with respective destination protocols of one or more destination applications; identifying the one or more destination applications; transporting the data having the technology independent form; transforming the data from the technology independent form to a technology dependent form associated with the respective destination protocols of the one or more of the destination applications; and transmitting all or a portion of the data in the technology dependent form to the one or more of the destination applications.

11. The method of claim 10, further comprising processing the data in the technology independent form.

28. A system for transporting data among applications, comprising: an input data adapter comprising an input interface and an input data converter, the input interface receiving an input data stream comprising informational content and having a technology dependent form associated with a source protocol of a source application, the input data converter converting the input data stream having the technology dependent form to input data having a technology independent form not associated with the source protocol and not associated with a plurality of destination applications; a processor communicatively coupled to the input adapter and coordinating the input data having the technology independent form, the processor coordinating transmission of all or a portion of the input data to the plurality of destination applications; and a plurality of output adapters each communicatively coupled to the processor and a respective one of the plurality of destination applications, each of the output adapters comprising an output data converter that converts the input data having the technology independent form to an output data stream having a technology dependent form associated with a destination protocol compatible with a respective destination application, and each of the output adapters further comprising an output interface that transmits the output data stream to the respective destination application.

30. The system of claim 28, wherein each of the output data adapters implements logic for processing the input data having the technology independent form.

34. The system of claim 28, wherein the processor is coupled to a receive queue and a plurality of send queues, the receive queue receiving the input data having the technology independent form from the input data adapter and the processor coordinating transmission of all or a portion of the input data having the technology independent form to one or more of the send queues.

35. The system of claim 34, wherein the processor communicates control signals to the send queues to coordinate transmission of all or a portion of the input data having the technology independent form to one or more of the output data adapters.

36. The system of claim 35, wherein processor coordinates transmission of the input data having the technology independent form to one or more of the output data adapters in an asynchronous or pseudo-synchronous manner.

38. A system for transporting data among applications, comprising: a plurality of input data adapters each comprising an Input interface and an input data converter, each of the input interfaces receiving an input data stream comprising informational content and having a technology dependent form associated with a source protocol of a respective source application, the input data converters converting the input data streams having technology dependent forms to input data streams having technology independent forms not associated with the respective source protocols and not associated with a plurality of destination applications; a processor communicatively coupled to the input adapters and coordinating the input data streams having the technology independent form, the processor coordinating transmission of all or a portion of the input data streams having the technology independent form to the plurality of destination applications; and a plurality of output adapters each communicatively coupled to the processor and a respective one of the plurality of destination applications, each of the output adapters comprising an output data converter that converts a respective input data stream having the technology independent form to an output data stream having a technology dependent form associated with a destination protocol compatible with a respective destination application, and further comprising an output interface that transmits the output data stream to the respective destination application.

42. The system of claim 38, wherein the processor is coupled to a receive queue and a plurality of send queues, the receive queue receiving the input data streams from the input data adapters and the processor coordinating transmission of all or a portion of the input data streams having technology independent forms to the send queues.

43. The system of claim 42, wherein the processor communicates control signals to the send queues to coordinate transmission of the input data streams having technology independent forms to one or more of the output data adapters in an asynchronous or pseudo-synchronous manner.

44. A computer readable medium tangibly embodying a program executable for transporting data, comprising: receiving, from a source application, data comprising informational content in a technology dependent form associated with a source protocol; converting the data from the technology dependent form associated with the source application to a technology independent form not associated with the source protocol and not associated with destination protocols associated with one or more destination applications; identifying the one or more of the destination applications; transporting the data having the technology independent form; transforming the data from the technology independent form to a technology dependent form comprising a destination protocol associated with each of the one or more of the destination applications; and transmitting all or a portion of the data in the technology dependent form to the one or more of the destination applications.

51. A system for transporting data, comprising: means for receiving data comprising

informational content in a technology dependent form associated with a sou protocol from a source application; means for converting the data from the technology dependent form to a technology independent form not associated with source protocol and not associated with destination protocols associated with on or more destination applications; means for identifying the one or more destinatio applications; means for transporting the data having the technology independent form; means for transforming the data from the technology independent form to a technology dependent form comprising a destination protocol associated with each of the one or more of the destination applications; and means for transmitting all or a portion of the data in the technology dependent form to the one or more of the destination applications.

L6: Entry 1 of 1 File: USPT Sep 17, 2002

DOCUMENT-IDENTIFIER: US 6453356 B1
TITLE: Data exchange system and method

Detailed Description Text (76):
In general, each Common Object is given a unique Object Identifier or OID so that any child or related objects can be associated with each other. All objects created as a result of this original OID require that this initial OID be stored as part of the object, regardless of whether the new object is a direct child object or whether the original object still exists. If the original OID were not stored, it would not be possible to correlate response objects with the initial request object. The OID value is automatically set during instantiation. Parent OID values are updated automatically when AddAttribute( ) is invoked, including all Common Objects that are contained within a ListObject.

Detailed Description Text (112):
When initiated, a component creates an instance of a System Configuration Object (DX_SysConfigObject) that stores the current parameter settings. The component also registers for a Signal/Event so that it is informed of changes to the configuration using the dynamic configuration command line interface/web interface. When a user wants to change the run time parameters of a component (identified by the process ID and the machine on which it is running), a signal/event is sent to the component to update its configuration. A signal/event handler invokes the ReconfigParameters ( ) method on the DX_SysConfigObject, which takes care of reconfiguring the various controller objects, such as DX_TraceLogObject, DX_QueueManager, and DX_ThreadController for example. The System Configuration object, DX_SysConfigObject, is a singleton object that initializes and controls the configuration of a component in the data exchange system, such as logging/tracing levels, thread controller, queuing, and performance monitoring. A singleton object, as understood in the art, refers to a C++ nomenclature meaning that only a single instance of the class may exist within a single executable. A singleton object is most commonly used when controlling system wide resources.

Detailed Description Text (155):
Changes in the run-time monitor configuration are handled through the system configuration utility. The System Configuration Object contains an instance of Monitor Object which is initialized with the monitor configuration parameter values specified in the System Configuration file and saves them as configuration member data. When the user modifies the Monitor Configuration parameters at run time, typically be use of the deconfigset command, a signal is sent to the component which calls the method ReconfigParameters( ) on the System Configuration Object to re-initialize the parameters. This method, in turn, calls reconfigparameters( ) on the Monitor Object and updates the configuration member data. The monitor thread reads the configuration member data when it becomes active in the next time interval. As such, the monitor configuration parameters are modifiable at run-time.

Detailed Description Text (173):
The DX_QMonitor utility informs the change in the audit logging status of a queue

to all the processes using the Queue by updating the registration file and raising an event/signal to inform all the components to update their DX_QueueLogMgr objects. This command is also responsible for cleaning up entries in the registration file corresponding to components that died without cleaning up the registration file. Whenever an Enqueue or Dequeue operation is committed, a check is made on the DX_QueueLogMgr to see if audit logging is ON or OFF and information is logged in case it is ON. When an instance is terminating, the destructor of DX_QueueLogMgr should update all the registration files in which it has created an entry. An implementation example is provided as follows:

48/31-34

Detailed Description Text (177):
Concerning data exchange system security, the basic security control is focused on the queue files access or the database tables access. The file access control requires the application user to be in a specific user group. The user group should be set before the application runs. The database table access control requires the application users to have the correct user name and user password. The user name and user password may be set in the environment variables or be hard coded in the application programs. In one embodiment, all applications share one database user account. This user account has privileges to create/update/delete tables in the view.

Detailed Description Text (201):
Since the external API is limited to the Enqueue/Dequeue API, the only limit to multiple access is the row-level table locking that the database supports. The file based queue mechanism uses simple file record-lock to protect from multiple updates to the file from multiple threads. The queue access operations for file-based implementation are thread-safe, such that all the operations are mutex protected.

Detailed Description Paragraph Table (3):
class: DX_CommonObject: public DX_CommonBase { RWDECLARE_COLLECTABLE (DX_CommonObject); friend class DX_ListObject; friend class DX_FileSubQueue; public: virtual.about.DX_CommonObject(); // Constructors /**********/ //I name==NULL, the name is set to "NOT_SET". //If name is ""or contains a "." it will be set to "INVALID_NAME" /**********/ DX_CommonObject(const char* name=0); /**********/ //Creat a copy of an entire DX_CommonObject, //but with it's own unique OID value /**********/ DX_CommonObject* Clone (); /**********/ //All AddAttribute and AddPvtAttribute methods take ownership of //the pointers passed in to them. Do NOT delete the pointers after //a call to AddAttribute and AddPvtAttribute. The pointers will be //deleted when the container DX_CommonObject or DX_ListObject is deleted. // //NOTE: When using the following two methods for creating a DX_STRING attribute // // AddAttribute(const char* name, const int type, const void* value) // and AddPvtAttribute(const char* name, const int type, const void* value) // // the value is defaulted to be of type const char* and not UNICHAR* // Misuse will lead to unexpected behavior. /**********/ EreturnCodes AddAttribute(DX_CommonAttribute* newAttr); EreturnCodes AddAttribute (DX_ListObject* newAttr); EreturnCodes AddAttribute(DX_CommonObject* newAttr); EreturnCodes AddAttribute(const char* name, const int type, const void* value); EreturnCodes AddPvtAttribute(DX_CommonAttribute* newAttr); EreturnCodes AddPvtAttribute(DX_ListObject* newAttr); EreturnCodes AddPvtAttribute (DX_CommonObject* newAttr); EreturnCodes AddPvtAttribute(const char* name, const int type, const void* value); /**********/ //Delete and DeletePvtAttribute will remove the named attribute //from the container

DX_CommonObject or DX_ListObject and delete the named //attribute's pointer /***/ EreturnCodes DeleteAttribute(const char* name); EreturnCodes DeletePvtAttribute (const char* name); /***/ //RemoveA will remove the named attribute from the container //DX_CommonObject or DX_ListObject but will not delete the named //attribute's pointer /***/ EreturnCodes RemoveAttribute(const char* name); EreturnCodes RemovePvtAttribute (const char* name); /***/ //Do NOT delete the pointer that is returned to you, it still is owned by //the container DX_CommonObject or DX_ListObject. You CAN use the any of //the attribute class methods for the pointer. /***/ void* GetAttribute(const char* name); void* GetPvtAttribute(const char* name); void PrintContents(); //The caller of Demarshal is responsible for object's memory allocation static DX_CommonObject* Demarshal(char* ObjOid, int type, int ContextIndex); EreturnCodes Marshal(int type, int ContextIndex); static EreturnCodes DeleteStorage(const char* oidVal, int type, int ContextIndex); //The caller of GetOID is responsible for deleting the memory of the returned char* char* GetOID(); EPriorityCode GetPriority(); protected: void* Find(const char* name); static EreturnCodes RestorePersistentObject(const char* oidVal); static EreturnCodes DeletePersistentObject(const char* oidVal, int type); private: //Data Members DX_ListObject* PrivateList; DX_ListObject* PublicList; static void Delete (RWDlistCollectables* list); static int Copy(RWDlistCollectables* dest, const RWDlistCollectables *src); static EreturnCodes CopyPersistentObject(char* filename); static EreturnCodes CopyFile(char* filename,char* copyfilename); void saveGuts(RWFile& file) const; void saveGuts(RWvostream& stream) const; void restoreGuts(RWFile& file); void restoreGuts(RWvistream& stream); const int check() const; DX_Boolean updateListOids(DX_ListObject* Plist, DX_OID *Poid); DX_Boolean updateObjectOids(DX_CommonObject* Pobj, DX_OID *Poid); DX_Boolean ValidateNaming (const char* name); #ifdefDX_DATABASE EreturnCodes InsertIntoTable(char* filename, int ContextIndex); static EreturnCodes RetrieveFromTable(char* filename, char* oid, int ContextIndex); EreturnCodes UpdateTable(char* filename, int ContextIndex); #endif //use to cal. the number of bytes needed to store an object using RWFile RWspace binaryStoreSize() const;}; };

Detailed Description Paragraph Table (34):
class: DX_QueueLogMgr { public: // create an instance of the object by calling the constructor // delete using DeleteInstance( ) static DX_QueueLogMgr* Instance(char* ComponentName=0); // delete the object static void DeleteInstance( ); // Checks if the QueueStatusList has an entry // for the particular queue and if so checks // if the entry indicates whether the // queue monitoring is on or off and logs the // QueueObject accordingly static void DumpQLog(EQueueOperation op, char* queueName, DX_QueueObject* qo); // if you want to monitor a queue, an entry in // the queue status list should be created first // by giving the queue name and the pid. The function // reads the .reg file and initializes the entry // accordingly. EreturnCodes InsertQueueStatusList(char* QName, int pid); // if queue monitor changed the queue registration file, reset the // object, read the reg file and update the queueStatusList; EreturnCodes ReconfigQueueStatusList( ); private: DX_Mutex* listLock; static DX_QueueLogMgr* instance; char ComponentName [MAX_NAME_LEN]; char QlogDir[MAX_FILE_NAME]; DX_QueueLogMgr(char* inComponentName=0,char* qlogdir=0); virtual.about.DX_QueueLogMgr( ); EreturnCodes DeleteFromRegFiles(char* ComponentName,int pid); DX_INDICATOR GetStatusFromRegFile (char* regFileName, int& regFileExists); char* GetQRegFileName(char* QName); EreturnCodes FindQueueLogStatusInfo(char* inQName, DX_QueueLogStatusInfo** copy); RWDlistCollectables *queueStatusList; }; class DX_QueueLogStatusInfo: public RWCollectable { friend class DX_QueueLogMgr; private: char qName[MAX_NAME_LEN]; DX_INDICATOR qLogStatus; char qLogFileName[MAX_FILE_NAME]; RWDECLARE_COLLECTABLE

(DX_QueueLogStatusInfo); DX_QueueLogStatusInfo( ){ }; // During construction if inQLogStatus is ON // open the logStream for QName.qlog in directory DX_HOME/DX_QLOG DX_QueueLogStatusInfo(char* inQName, DX_INDICATOR inQLogStatus, char* qlogdir); // close all the streams which are open .about.DX_QueueLogStatusInfo( ); char* GetQName( ); DX_INDICATOR GetQLogStatus ( ); char* GetQLogFileName( ); // if set to on, open stream, if set to off close the stream EreturnCodes SetQLogStatus(DX_INDICATOR inStat); }; #define DX_QLOG (QueueOp,QueueName,qo) DX_QueueLogMgr::.backslash. DumpQLog(QueueOp,QueueName,qo);

Detailed Description Paragraph Table (43):
class: DX_QueueManager { friend class DX_QueueTransaction; friend class DX_DBQueue; friend class DX_Monitor; public: static DX_QueueManager* Instance(const char* ProcessName, EstorageTypes type); static void DeleteInstance( ); static DX_QueueManager* GetInstance ( ); /*****/ //Queue operation interface /*****/ /****** label and comment arguments will be used for Queue Administration Purpose. //So queue administration GUI will also see the name and comment of each CO in the //queue. They can be type of UTF-8 encoded string, 7-bit ASCII string or wide string. //User should not delete pointer to DX_CommonObject after call Enqueue /*****/ EreturnCodes Enqueue(const char* qName, DX_CommonObject &co, DX_QueueTransaction &transaction, const char* oLabel, const char* comment, EPriorityCode priority = NORMAL); EreturnCodes Enqueue(const char* qName, DX_CommonObject &co, DX_QueueTransaction &transaction, const UNICHAR* oLabel, const UNICHAR* comment, EPriorityCode priority = NORMAL); /*****/ //Caller should free non-NULL pointer to DX_CommonObject //Return SUCCESS if Dequeue returned a common object //Return FAILED if Dequeue returned a NULL common object pointer /*****/ EreturnCodes Dequeue(const char* qName, DX_CommonObject* &CO DX_QueueTransaction &transaction); EreturnCodes Dequeue(const char* qName, const char* objID, const char* objLabel, DX_CommonObject* &co, DX_QueueTransaction &transaction); /*****/ // caller of GetCursor is responsible for deleting //returned DX_IndexObject* pointer. /*****/ DX_IndexObject* GetCursor(const char* qName, EPriorityCode priority = INTERACTIVE); /*****/ //W set the cursor to the EPriorityCode passed in /*****/ EreturnCodes ResetCursor(DX_IndexObject &cursor, EPriorityCode priority = INTERACTIVE); /*****/ //T always is a non-null DX_QueueList returned //Caller should delete DX_QueueObjectList returned // //NOTE: DX_QueueList may be empty if no entries were found // //USAGE: //GetQueueView(DX_QueueObjectList* &list DX_IndexObject &QViewCursor, // int size = 0) //will return #entries =< size for EPriorityCode of QViewCursor and ALL lower priorities // //GetQueueView(DX_QueueObjectList* &list, EPriorityCode priority, // DX_IndexObject &QViewCursor, int size = 0) //will return #entries =< size for EPriorityCode of priority, QViewCursor is updated to //reflect position of last retrieved entry. /*****/ EreturnCodes GetQueueView(DX_QueueObjectList* &list, DX_IndexObject &QViewCursor, int size = 0; EreturnCodes GetQueueView(DX_QueueObjectList* &list, EPriorityCode priority, DX_IndexObject &QViewCursor, int size = 0); /*****/ //Caller should free char** returned twice /*****/ char** GetManagedQueueNames(int& number); char** GetAllQueueNames(int& number); private: static DX_QueueManager* instance; char* owner; EstorageTypes Implementation; char*

FileDBDirectory; RWGDlist(DX_Queue) QueueList; DX_Mutex* mutex; DX_QueueManager (const char* processID, EstorageTypes type, const char* FileDBDir); .about.DX_QueueManager( ); //Extended transaction support interface EreturnCodes Commit(DX_QueueTransaction &transaction); EreturnCodes Rollback(DX_QueueTransaction &transaction); //Queue administration interface EreturnCodes DeleteQueue(const char* qName); EreturnCodes FlushQueue(const char* qName); EreturnCodes FlushQueue (const char* qName, EPriorityCode priority); EreturnCodes RemoveFromQueue (DX_QueueObject *object); //Performance monitor interface //the memory of pNumOfMsgProcessed, pAvgMsgCacheTime, pAvgMsgProcessTime //should be allocated before invoking this method. EreturnCodes GetQueuePerformance(char * inputQName, long *pNumOfMsgProcessed, double *pAvgMsgCacheTime, double *pAvgMsgProcessTime, DX_Boolean resetFlag = TRUE); void ResetAll(const char *qNameList); EreturnCodes GetDBTableSpaceUsage(float &usage); //Internal use DX_Queue* FindQueue(const char* qName); DX_Queue* CreateQueue(const char* qName); };

Detailed Description Paragraph Table (46):
class: DX_FileSubQueue { friend class DX_FiIeQueue; friend class DX_QueueManager; private: DX_FileSubQueue(const char* qName, EPriorityCode pCode, const char* FileDBDir); .about.DX_FileSubQueue( ); EreturnCodes Enqueue(DX_CommonObject& co, const char* ProcessId, const char* label, const char* comment); DX_CommonObject* Dequeue(DX_IndexObject& io); DX_CommonObject* Dequeue(DX_IndexObject& io, const char* ObjectID, const char* ObjectLabel); EreturnCodes Commit(DX_QueueOperation& oper); EreturnCodes CompleteCommit(DX_QueueOperation& oper); EreturnCodes Rollback (DX_QueueOperation& oper); EreturnCodes DestroyStorage( ); EreturnCodes Flush( ); EreturnCodes RemoveObject(DX_QueueObject* qObj); EreturnCodes GotoBeginning (DX_IndexObject& cursor); EreturnCodes GetQueueView(DX_IndexObject &cursor, DX_QueueObjectList* &list, int size); //used by the performance monitor long GetNumOfMsgProcessed( ); double GetTotalMsgCacheTime( ); double GetTotalMsgProcessTime( ); void Reset( ); private: char* QueueName; EPriorityCode Priority; char* QueueFileName; char* IndexDirectory; int startFileIndex; int endFileIndex; //These two fields are used for Dequeue operation and always go forward int currentFileIndex; int currentRecordIndex; int lastRecordIndex; DX_QueueObjectList BufferList; ////////////////////////// //Internal use only ////////////////////////// EreturnCodes EnqueueCommit(DX_QueueOperation &oper); EreturnCodes DequeueCommit(const char* oid, const DX_IndexObject* io); EreturnCodes CompleteEnqueueCommit(DX_QueueOperation &oper); EreturnCodes CompleteDequeueCommit(const char* oid, const DX_IndexObject* io); EreturnCodes EnqueueRollback(const char* oid, const DX_IndexObject* io); EreturnCodes DequeueRollback(const char* oid, const DX_IndexObject* io); EreturnCodes TryRecycleFile(int fIndex); void TryRecycleQueue( ); char* GetFileName(int fIndex); int OpenFile(int whichFile); EreturnCodes CreatEndFile( ); void UpdateIndexFile( ); void UpdateFileIndex( ); EreturnCodes ExpandQueue( ); EreturnCodes MarkObjectInFile (const DX_IndexObject* iObj, EQueueObjectStatus status); //Unix does not have low-level eof IO function available int IsEOF(int fd); //used by performance monitor double totalMsgCacheTime; double totalMsgProcessTime; long numOfMsgCommited; void SetDequeueTime(DX_QueueObject *qo); void SetEnqueueTime(DX_QueueObject *qo); void CompleteCommitCalculation(DX_QueueObject *qo); };

Other Reference Publication (3):
Greenbaum, J., "Competitve Linking Update: CrossRoads to the Rescue," Hurwitz Balanced View Research Bulletin, 6 pgs. (Jun. 1997).